The Os++

User's Manual

THOMAS RICHTER

Contents

1 Overview on the Second Edition

The second edition of Os++ fixes a couple of bugs and improves some of its features. Most notably, the editor had a bug by running into a `BRK` instruction when erasing the cursor, which was harmless with the Os++ default interrupt handler installed, and FMS++ did not handle "boot acccess" open modes with AUX2 equal to 128 properly. These defects have been fixed. The FMS++ overlay manager now also supports banking the FMS buffers under the Os ROM, reaching MEMLO values of `$800`. Previous versions could only FMS buffers under Basic or an Oss switching cartridge.

2 Overview on Os++

Os++ is an open source replacement operating system for the Atari 8-bit series of home computers. Even though Os++ was originally developed as part of the Atari++ emulator project, the operating system is stand-alone and will also work on a real machine, unaltered.

Os++ requires at least an Atari 600XL (extended or native), or an 800XL, or any newer model. Upgraded older systems will work as long as memory bank switching is supported correctly. In specific, Os++ requires banking support for the *Self Test* memory window to be functional, even though it serves another purpose here.

Os++ consists of two parts: A resident ROM image, supplied with this software package, that replaces the original 16K ROM built into a machine. This ROM is completely sufficient to operate the system, and it provides many more features the Atari ROMs did not supply, discussed in the next paragraphs. The second part is the Os++ *System Disk* that includes additional tools and utilities, very much in the spirit of the Atari system disks that came with the 810 or 1050 disk stations.

The design goal of Os++ was to provide an open source operating system for the 8-bit machines that supplies all necessary functionality for the all-day use of the 8-bit series as it was shipped by Atari. This means that the development of Os++ was guided by the hardware available at the time the machines were originally sold, and to supply the best possible solution — as ROM space permits — at this particular time. This means, of course, that several compromises had to be made, and in particular, that hardware that appeared later by third-party vendors could not be supported.

In the author's opinion, one of the largest drawbacks of the original Os design was the lack of a built-in file management system many competing home computer systems offered. While the Atari could and did bootstrap its DOS — the Atari name for the file management system plus utility package — from disk, the bootstrap process was lengthy, and also took precious RAM space. Os++ supplies a ROM-based FMS that is backwards compatible to Dos 2.0S and Dos 2.5, the most popular Dos versions back at the time of the 8-bit series was sold, and enables thus to use the 1050 disk station even without booting from them. Nevertheless, bootstrapping custom file management systems, or games, still remains available, the ROM will not get into their way. FMS++, the file management system in the ROM, improves the original Atari design in many ways, and provides a couple of useful features, as for example taking as little as 256 bytes of precious BASIC RAM-space. The FMS is described in section 8.

Many errors found in the original Atari ROMs are not present in Os++[1], and a couple of improvements have been made. The editor and screen handler have been improved. The math pack is both faster and more precise. The resident disk support includes additional commands.

Needless to say, with fitting more features into the same ROM space, some things just had to go, and the author tried to find compromises of what could go and what should not. You may or may not agree with these goals, but as Os++ is open source, you may probably contribute and improve any future release to what *you* consider important.

The resident tape handler for the Atari program recorders is no longer available. Instead, it has been replaced by a RAM-based handler that can optionally be loaded from disk if required. It was considered

[1] Even though the author is quite confident that probably *other* errors, but hopefully less of them, are in Os++

by the author that, even back then at the high-peak of the Atari 8-bit series, tapes were already an outdated technology that only allowed unreliable storage of data; their usage was annoying, and slow. The new disk-based tape handler, described in section 15, offers the same functionality the ROM based original tape handler did, but improves upon its features by implementing a faster *Turbo* mode that improves recording speed a little bit.

Furthermore, the parallel-port support of the original Atari ROM is no longer present, for several reasons. First of all, the author had the feeling that Atari's only half-hearted support was insufficient to create enough market impact to make this idea successful. Even Atari's later designs, for example the 80-column extension, did not make use of it. Finally, third-party hardware became available after Atari stopped supporting the 8-bit series at all, much too late. Thus, its popularity was rather limited. Second, and as a completely pragmatic reason, the author found that the available documentation and lack of own hardware made it unreasonable to implement it. Third, the ROM space was needed for better purposes. It is nowadays probably easier to emulate external disk drives by PCs, such that parallel port extensions are not needed in first place.

This manual is structured as follows: The next sections will discuss the extensions made in the ROM, both from the perspective of the BASIC programmer, as well as from the perspective of an advanced user. This is followed by a somewhat longer description of FMS++, the ROM-based filing system, which is an entirely new feature. The ROM-based DOS command prompt is explained next. This command prompt is quite similar to the Os/A+ product series supplied by Oss as third-party extension, though resides in ROM. The next sections describe the tools on the System Disk: The FMS overlay manager, the RS232 bootstrap, the DiskIO BASIC extension and the non-resident DOS tools. An entire chapter is devoted to the disk-based, menu driven Disk Utility Package. This package is similar in nature to the DUP menus supplied by Atari, but is both feature-rich, very modular and user-configurable. The last section describes the disk-based tape handler. The manual includes an Annex on the additional ROM-based entry points, and on changes in the memory map.

Thomas Richter, Berlin, April 2020

3 Booting Os++

Since Os++ features a ROM-based file management system, the disk drive need not to be turned on or loaded with a disk. If turned on, the disk handler will be already present and available for use, regardless of whether the system disk was available or not.

If you want to run BASIC, just turn the system on, and either insert the system disk, or leave the disk drive empty. If the system disk is available, additional tools described in sections 8 and 9 will be installed, but usage of these tools is optional and not required. For example, the FMS++ overlay manager will reduce the amount of BASIC RAM required for the filing system to a minimum, and the DiskIO program will enlarge Atari BASIC by disk-related commands in its direct mode.

If you want to play a game, insert the game disk in drive 1, turn the computer off, press the `OPTION` key and hold it down, and turn turn the machine on while holding `OPTION`. This will disable BASIC and boot from disk.

If you want to use a custom DOS, and do not plan to use the ROM-based filing system, insert the corresponding DOS disk into the drive, then turn the machine off, and on again. The system will then boot from disk as it ever did, and the ROM-based FMS will not get into the way. If you also hold option while turning the machine on, BASIC will be disabled.

To return to the built-in FMS from BASIC, and disable any DOS loaded from disk, type the command `BYE` into the BASIC screen and press `RETURN`.

You *can no longer* boot from tape by pressing the `START` key as the tape handler is no longer in ROM. However, tape booting is still supported by the disk-based tape handler. How this works is described in section 15.

4 The Editor Handler

The editor handler is the part of the operating system that provides the text display and text entry, where BASIC programs are entered and their textual output is printed. The editor handler is also responsible for printing text in the text windows in the dot-graphic modes of the machine, and collecting input from the user in these windows. The text editor is the part of the operating system that is responsible for displaying the `READY` prompt the machine displays when turned on.

The Os++ editor does not differ significantly from the original editor handler, it only features one single enhancement: If you insert blank-spaces into a line with `CONTROL+Insert`, the editor will run the console buzzer as soon as the line gets very long and you run into the risk of moving characters out of the line. As in the original editor handler, such characters are then lost, but the original editor handler did not warn in such a situation.

4.1 Information for BASIC Users

As in the original Atari ROM, the editor handler has the device name `E`. It supports the open modes 4,8,12 and 13, where the latter reads continuously lines from the screen without waiting for the `RETURN` key. The open mode is specified as the second argument of the BASIC `OPEN` command. For example,

```
OPEN #1,12,0,"E:"
```

opens channel 1 to the editor. The mode 12 implies that reading and writing to the editor is permitted. Mode 4 is read-only, mode 8 is write-only, and mode 13 is special and was already described above.

Channel 0 is already opened by the operating system, and always talks to the editor.

4.2 Information for Advanced Users

The Os++ editor device is also improved in other ways less visible to the user. The original editor handler erroneously wrote to data beyond the screen end when inserting lines. This bug no longer exists in Os++. Furthermore, the editor and the screen handler — see next section — have been largely decoupled.

Smooth-scrolling remains available, and can be turned on by first setting the RAM-location 622 (hex $26e) to a value larger or equal 128, and then re-opening the editor. Unlike the original Atari ROMs, smooth scrolling also works in the text windows when the editor shares the screen with the screen handler.

5 The Screen Handler

The screen handler is responsible for creating graphical display modes, text modes with larger or more colorful characters, and printing and plotting on the screen. All the original commands and screen modes of the Atari screen handler are supported, but unlike the Atari handler, the graphic modes 9,10 and 11, making use of special modes of the GTIA chip, are also available with a text window. Reading from and writing to the screen handler is a little bit different from the original Atari handler in some ways that should not impact compatibility. These differences should not bother users of BASIC as all BASIC functionalities that concern the screen handler, for example `PLOT`, `DRAWTO` or `LOCATE` continue to work. They may make a difference for the advanced user. Such changes are discussed below.

5.1 Information for BASIC Users

The screen handler is addressed by either opening a channel to the `S` device, or by using the BASIC `GRAPHICS` command, both ways are mostly equivalent, though the latter is more convenient. For this reason, only the latter will be discussed here.

One extension of the screen handler concerns the modes 9,10 and 11. Unlike all other modes, these three special modes *do not* create a text window by default. That is, while

```
GRAPHICS 15
```

will open a 160×160 four-color screen with a four-line text window at its bottom,

```
GRAPHICS 11
```

will not. In fact, if entered in direct mode, only the screen will be cleared. This is because the editor handler BASIC uses for its input notices that it does not have a text window for reading its input from, and thus the usual editor screen is re-opened immediately afterwards. Thus, this mode was only available by program code, but not by typing commands into the console in direct mode.

With Os++, you can, however, force a text window even in modes 9,10 and 11. For this, add 64 to the graphics mode:

```
GRAPHICS 11+64
```

will open a 80×160 screen with 16 colors and a four-line text window.

For backwards compatibility,

```
GRAPHICS 0
```

opens the default text window and makes it available for the editor. The same happened with the Atari ROMs. Without further tricks, a text window is *not* available in this mode and the full screen will be used by the editor then.

5.2 Information for Advanced Users

The original screen handler was quite "intertwined" with the editor handler, which had some unpleasant side effects. For example, one could *print* to a screen screen by means of

```
PRINT #6;"HELLO WORLD!"
```

and the screen would scroll if 24 lines were filled, regardless of whether the screen was actually 24 lines high or not. Actually, the above could even scroll graphic modes, albeit only partially.

Printing to the screen of course also works with Os++, but the screen never scrolls. Scrolling is done by the editor device only, and not by the screen handler. If printing reaches the bottom of the screen, an error 141 *Out of Screen* will be generated.

This has the side-effect that the screen device behaves pretty much line a *frame buffer device*. The entire screen can be filled with graphics by first positioning the cursor in the top left, and then writing data to it. The reverse also goes for *reading* from the screen: This will read data sequentially from left to right, top to bottom, until the end of the screen is reached. This data could then, in principle, be saved as a screen dump, by a single CIO command.

Otherwise, the screen handler supports all operations the Atari implementation supported, namely plotting points, the line drawing algorithm, and the simple right-fill algorithm. For reasons of backwards compatibility, the special ATASCII codes 125 (hex $7D) and 155 (hex $9B) will clear the screen, and advance the graphics cursor to the next line. Actually, these two functions are part of the editor functionality and should not go into the screen handler, and rather should have been replaced by corresponding CIO commands, but have not to keep legacy software operational. All other ATASCII codes are not interpreted and directly specify glyphs or color to be printed on the screen.

As with the original ROMs, the graphics mode 0 is special and actually opens an *editor* screen, and not a channel to the screen handler. This unorthogonal design goes back to Atari and is reproduced here. One can force a four-line text window in this mode by setting the RAM variable 703 (hex $2BF) to 4. The window is turned off by setting the same variable to 24, the height of the full screen. The Os++ screen handler would also support other values here, but this requires a careful preparation of the screen and is not discussed here nor currently officially supported.

6 The Keyboard Handler

The keyboard handler reads input from the keyboard, but unlike the editor, it does not echo the printed characters on the screen. Obviously, the keyboard handler can only read data, and not write data. The device name of the keyboard handler is `K`.

The keyboard handler in Os++ does not differ significantly from the one in the Atari ROMs. The only difference is that the function keys `F1` through `F4` that were present on some prototypes of the XL computer series are not supported, simply on the basis none of the follow-up mass-produced models have them.

6.1 Information for BASIC Users

An I/O channel to the keyboard device is opened in the same way as for the Atari ROMs. The open mode has always to be 4, indicating read-only access:

```
OPEN #1,4,0,"K:"
```

Once the channel is open, a program can wait for keyboard input by means of the `GET` instruction. For example, once a channel to the keyboard handler has been established by the above line, the following instruction

```
GET #1,A
```

will wait for the user to press any key on the keyboard, and return the ATASCII code of the key in the BASIC variable A. No output will appear on the screen.

As under the original ROMs, the keyboard handler does not provide a function to *test* whether a key is actually pressed. It will always wait for the user and thus halt the program.

6.2 Information for Advanced Users

Similar to the Atari XL ROMs, the keyboard map can be redefined by installing a pointer to the keyboard definition table at the RAM addresses 121 and 122 (hex $79, $7A). In addition to regular ATASCII codes, extended editor commands are also supported. These commands are only understood by the editor device and do not return ATASCII codes. They can move the cursor to the home or end position, toggle the keyboard click and various other functions. The only difference is that the function key definition table is no longer available.

Furthermore, the keyboard click generator does not use the ANTIC WSYNC register to time the delay loop defining the keyboard click frequency. Thus, the keyboard click does not interfere with display list interrupts or other system services that depend on precise timing.

7 The Printer Handler

Similar to the Atari ROMs, Os++ includes a built-in printer handler that talks to printer devices connected to the SIO bus, either directly, or indirectly by an interface box. The printer handler is functionally identical to the one included in the Atari ROMs. The device name of the printer handler is `P`.

7.1 Information for BASIC Users

The printer can only be opened for writing, and thus the open mode eight must be used. The secondary open mode specifies the number of characters that fit into the printer buffer, though in a non-obvious way. The printer buffer keeps the characters printed from BASIC until either an end of line character is printed or the buffer runs full. Only then all buffered characters are forwarded to the printer; thus printing characters to the `P` device might not have an immediate result. This keeps the communication to the printer "relatively" fast.

The following command opens a channel to the printer:

```
OPEN #1,8,A,"P:"
```

where A can take the following values: If A equals 68, the ATASCII code of the letter `D`, the printer buffer is 20 characters long. This indicates *double width* printing. If A equals 83, the ATASCII code of the letter `S`, the printer buffer becomes 29 characters long; this should initiate *sideways* printing. In all other cases, and specifically for the case `A=78` for ATASCII `N`, the buffer is 40 characters long and normal printing is initiated. Whether the printer actually does print sideways or expanded to double width is, however, another matter. Most printers will simply ignore these instructions, and will instead require proprietary control sequences to change the printing style. The description of these sequences is beyond the purpose of this manual.

To actually print text, the `PRINT #` command of BASIC should be used. For example, following the above `OPEN` instruction,

```
PRINT #1;"HELLO!";
```

will push the string "HELLO" into the printer buffer. No output will appear because the printer buffer will absorb the data. The data is finally printed when either an end of line is printed, the buffer runs full, or the channel is closed again. That is,

```
CLOSE #1
```

will finally make the text appear.

BASIC does also have a printer specific command that includes all these steps in one, namely opening a channel to the printer, printing the data, and closing the channel again. This is

```
LPRINT "HELLO";
```

However, usage of this command is discouraged since it does not respect the output formatting of the usual `PRINT` command. The semicolon, unlike the ordinary `PRINT`, will not stop the print head from advancing to the next line. This is, unfortunately, a side effect of the implicit `CLOSE` instruction within `LPRINT` which always generates an end of line to empty the printer buffer.

7.2 Information to Advanced Users

Actually, the printer handler can handle more than one printer, which are then addressed by `P1` through `P9`. However, only one printer can be handled at a time. If more than one channel is open to the printer at once, proper function of printing cannot be ensured. For example, it will happen that data buffered for one printer might appear on a different printer. Please open only one channel to the printer a time.

8 The Disk Handler

Unlike Atari operating systems, Os++ comes with a resident disk handler, also called a file management system, or short FMS. This system administrates disks, and allows to access data stored on disk by file names. The package of the file management system and additional disk utilities is also called the *DOS* (Disk Operating System) though the name is not quite appropriate. Os++ contains both the FMS, here denoted by FMS++, and a DOS command line in ROM. A more traditional menu-driven utility package is available on the system disk. The ROM-based command line is described in section 11, the menu in section 12. This section covers only the FMS.

Because the Atari ROMs did not provide a disk handler, the DOS had to be booted from disk, there taking both disk space, and also taking RAM space that should better be used for your programs. The option of booting a third party FMS still exists, however, and depends on whether a bootable disk is detected when the system is turned on. If so, it tries to run the program recorded in the boot sectors of the disk, which may — for example — also contain a file management system, then replacing the built-in one.

8.1 Features of FMS++

The ROM based file management system is derived from Dos 2.XL, a feature-rich DOS by the same author. It formats $5\frac{1}{4}$ disks on a 1050 floppy to 963 free sectors, each of which stores 128 bytes of data. The format is completely backwards compatible to Dos 2.0S, the first and most wide-spread DOS by Atari. You can safely read and write to Dos 2.0S disks and vice versa. FMS++ also supports reading Dos 2.5 disks, though is not able to write to sectors beyond the 720 sector barrier because Dos 2.5 administrates them in a way that is non-transparent to 2.0S. Dos 2.5 should not be used to write to FMS++ disks.

FMS++ offers two methods to access the disk: The *file mode* where the disk is structured in files, and each file can be accessed, read and written to as a separate entity. The *direct mode* allows accessing disk sectors directly, i.e. the entire disk then appears as one big file. The *direct mode* has the advantage that random access to data on the disk is much easier, but it is not compatible to the *file mode*, and the same access modes should not be used on the same disk. The *direct mode* is ideal for storing random access data, as from a database, on disk, but then each database requires its own disk. Direct mode is an advanced feature not suitable for beginners. It is a new access mode provided by FMS++ not available by any Atari DOS.

In the *file mode*, each disk contains a disk directory that keeps a catalog of the files on the disk. A catalog entry consists of the base name — eight letters or digits — followed by a three letter extension that, by

convention, classifies the purpose of the file. Additionally, each disk may have a *headline* or *disk identifier* that is only visible to FMS++. Neither DOS 2.0S nor DOS 2.5 will be able to print this name, but it will appear topmost in the directory when the disk contents is listed by FMS++.

When specifying a file name to the DOS, you need to provide first the name of the disk device handler, which is `D1` through `D8` for the first through eighths drive. Alternatively, `D` alone also addresses the first drive. The device handler is separated from the base name by a colon "`:`", which is again separated from the extension by a dot "`.`". A file name may be followed by modifiers, which are appended at the end with a forwards-slash "" and a letter or a digit. Several modifiers can be appended, then each of which must be introduced by a slash. So for example

```
D:PROGRAM.BAS
```

is the name of a file on the first disk drive, the base name of which is "PROGRAM" and the extension of which is "BAS". By convention, the "BAS" extension denotes ATARI BASIC programs, though you may pick the extensions as you like. Modifiers are extended features that are discussed in section 8.11 and will be introduced there.

8.2 Booting from FMS++

Unlike the Atari ROMs, FMS++ is ROM based and thus does not require a bootable disk to be present. If the disk in drive 1 follows the boot-disk specifications of Atari, legacy booting will be run and FMS++ will not be installed. This allows to use custom DOSes or boot programs such as games. Otherwise, if either no disk is present, the drive is turned off or the disk is a FMS++ boot disk, FMS++ will be installed, and will follow its own boot procedure.

8.3 Booting for BASIC

For BASIC users, it is most convenient to boot from the Os++ system disk as it contains many helpful features and tool programs that may aid you in programming the Atari. Thus, turn the main system off, turn drive one one, insert the system disk into drive one, and turn on the main system. After a short while, the BASIC prompt will appear. By default, the system disk installs the FMS overlay manager, see section 8, which will reduce the memory footprint of FMS++ to 256 bytes leaving most memory for your program. It will also install the DiskIO BASIC extension, discussed in section 9. Additionally, any RS232 interface handler, if present, will be loaded into RAM. The boot procedure can be modified, and the tools to be loaded can be defined by the user. This will be discussed in section 12.10.

If you can spare the RAM or do not need or want DiskIO, you can also leave the disk drive door open. FMS++ will then not attempt to load the additional tools and will bring you to the BASIC prompt immediately. Still, the disk drive is then accessible because FMS++ resides resident in ROM.

8.4 Advanced Information on the Boot Process

When FMS++ starts its own boot process, it looks for three special files on disk. These files are in the Atari binary load format, specified in section 14, and are brought to RAM by the resident loader.

First, FMS++ looks for a file named `CONFIG.SYS` on the first disk drive. If this file is present, FMS++ will load it into RAM and run it, and afterwards re-initializes itself. The purpose of the `CONFIG.SYS` file is to modify FMS++ RAM vectors to modify its configuration, for example to set the number of drives it can administrate, or to define the number of files it can open at once. These RAM vectors are described in section 21, and a convenient way how to create the configuration file is shown in section 12.10. The FMS Overlay Manager discussed in section 13 is an advanced FMS++ configuration program.

Second, FMS++ loads the file `HANDLERS.SYS` into RAM. The purpose of this file is to bring additional disk-based device handlers into memory. The Os++ system disk contains a `HANDLERS.SYS` file that loads

the RS232 interface software into RAM; optionally, the file can be replaced or extended by a handlers file that loads the disk-based tape handler, see section 15.

Third, FMS++ scans the disk for a file named `AUTORUN.SYS`. The purpose of this last file is to run user application programs, games and other miscellaneous programs. The Os++ system disk contains the BASIC extension DiskIO as `AUTORUN.SYS` file.

8.5 Loading and Saving Programs from BASIC

The disk handler provides a convenient way to load and save BASIC programs from and to disk. The following command saves the current BASIC program in memory to the disk, storing it for later. It receives the file name `TEST.BAS`, and will be listed as such in the disk directory.

```
SAVE "D:TEST.BAS"
```

The program can be restored later on, for example after the computer has been turned off for a while, by

```
LOAD "D:TEST.BAS"
```

The file name `TEST.BAS` identifies the file on the disk.

The `SAVE` and `LOAD` command operate on BASIC programs in their *tokenized* form, where every BASIC instruction has been replaced by a very compact short *token*. The format is not human readable, but binary, though very compact. Besides in tokenized format, BASIC programs can also be stored in clear-text. This is done by

```
LIST "D:TEST.LST"
```

which is, however, much slower than `SAVE` because the BASIC interpreter has first to convert the program back from its tokenized form into clear text.

The reverse of `LIST` is `ENTER` which reads tokenized programs back from disk. `ENTER`, unlike `LOAD`, does not erase the current program from memory, but rather merges it with the new program from disk, i.e. lines present on disk will replace the same lines in memory, lines present in the disk version and not present in memory will be added, and lines not available in the disk version stay intact. Usually, this creates a mess, and it is advisable to clear the program memory first by `NEW` before using `ENTER`.

```
ENTER "D:TEST.LST"
```

restores a LISTed program from disk. Note that I use here the convention to give LISTed programs the extension `.LST`.

8.6 File Operations in BASIC

BASIC not only allows you to load and save programs from disk, it also allows you to store your data on disk, and retrieve this data later. To this end, you first need to open an I/O channel to a file on disk by means of the `OPEN` command:

```
OPEN #1,4,0,"D:TAX.DAT"
```

opens channel 1 to the file `TAX.DAT` on disk for reading, which is indicated by the number `4` as second argument to the `OPEN` command. A single decimal number can be read from the file by means of the `INPUT #` instruction:

```
INPUT #1,A
```

This will read the next number from the file to which channel one was opened, and will place this number in the variable A. Finally,

```
CLOSE #1
```

closes the file again.

Writing data to disk uses `PRINT #` instead of `INPUT #`, and requires that the channel is open for writing, indicated by the number 8 as second argument to `OPEN`:

```
OPEN #1,8,0,"D:TAX.DAT"
```

will create a new file `TAX.DAT` on the disk in drive one. If such a file exists already, it will be erased first and the new file will replace it.

```
PRINT #1;A
```

will print the contents of variable A to the file, and

```
CLOSE #1
```

will close the stream again.

> *Don't loose your data!* Do *not* press `RESET` while any disk channel is open for writing, and always `CLOSE` disk channels before turning off the computer, the disk drive or changing the disk. Not following this advice will corrupt the disk structure on the disk, and you will likely never see your data again.

8.7 Advanced Modes for Open

The `OPEN` instruction takes two *mode parameters*, of which only the first one was partially covered. Despite the choices for reading and writing, the following additional open modes exist (as above, the second parameter remains 0 here):

Modes 6 and 7 are special in so far as they do not open an existing file on disk, but rather open a channel to the disk directory, showing the contents of the disk. Each file is there shown as one line, together with its status and its size. For backwards compatibility reasons, the format used in the directory listing is unfortunately different from the format used in file names: First, the device specification, e.g. `D:` is missing. Second, the first character is either an asterisk if the file is protected from overwriting, or a blank star for regular files. The second character is always a blank, except for the last line. The next 8 characters are the base name of the file, and the next 3 are the extender. Unlike in the file name, base name and extender are not separated by a dot "`.`", but by spaces such that all entries are aligned. The last part of a directory entry is the size of the file in sectors, and the last line of the entire directory list the number of free sectors that remains available.

The following BASIC program lists the contents of the directory in drive one:

```
10 OPEN #1,6,0,"D:*.*"
20 TRAP 70
30 DIM A$(20)
40 INPUT #1,A$
50 PRINT A$
60 GOTO 40
70 CLOSE #1
```

The `TRAP` instruction is here used to capture the error if the `INPUT #` instruction hits the end of the directory. If you enter and run this program, you will see that the last line lists the number of remaining free sectors of the disk. The meaning of the `*.*` file name will be explained in section 8.9.

It is certainly inconvenient to have to type in this program every time the directory contents shall be listed. Instead, it is suggested to either go to the DOS command line by means of the DOS instruction, and type in DIR there, or install the DiskIO BASIC extension and use the DIR command of DiskIO at the BASIC prompt to bring the directory to the screen. The DOS command line is described in section 11, DiskIO is explained in section 9. Alternatively, Basic++ by the same author also includes a DIR command to print the directory.

Table 1: Open Modes

Mode parameter	Purpose
4	Open for reading only. The file will be opened for read operations only. Any write operation triggers an error. If the file does not exist, OPEN will fail.
6	Open the disk directory for reading. Write operations are not possible.
7	Open the disk directory, but hide the disk headline. Otherwise, this mode is identical to mode 6.
8	Open a file for writing. If the file already exists, it will be erased first and will be overwritten. Otherwise, the file will be created. Reading is not possible.
9	Open a file for appending. If the file does not yet exist, an error will be triggered. Otherwise, any data written will be appended at the end of the file.
12	Open a file for reading and writing. If the file does not yet exist, an error will be triggered. Otherwise, can be read sequentially, or old data in the file will be overwritten. Read and write operations start from the beginning of the file. Extending the file beyond its file end is not possible and triggers an error.
13	Open a file for reading, writing and extending. Similar to mode 12, except that the file will be automatically extended if a write operation crosses the end of the file.

8.8 Advanced Information on the Open Modes

Mode 7 is new to FMS++, it only suppresses the output of the headline in the directory listing and might be used to avoid compatibility issues with programs that cannot handle this extension.

Mode 9, the append mode, has also been extended. Unlike in DOS 2.0S or DOS 2.5, it never wastes disk space and appends right at the end of a file. Its internal operation is also somewhat different because, unlike the Atari DOSes, the OPEN operation now takes longer time, but CLOSE is rather immediate. The situation for the Atari DOSes was quite the reverse, OPEN worked rather quickly, but CLOSE took some time to re-establish the file link.

Mode 13 is just another extension and is very versatile as it also allows in some situations a quicker append to files than possible with mode 9. In such an application, a file would be first opened for writing, and before closing the file, the file pointer would be recorded by NOTE, see section 8.16. To extend then the file, a smart program would, instead of using mode 9, use mode 13, and set the file pointer right to the end of the file using POINT. Continuing to write will then extend the file. The DOS command line and the menu driven utility package make use of this trick to implement a better performing file copy.

Modes 8 and 9 are also extended in so far as a `POINT` command can reset the file pointer to the start — or any intermediate — position of the file. Continuing to write data will then overwrite the already existing data, as in mode 13. However, reading data is not possible in either of these modes.

While mode 6 is certainly not new, it allows several channels to be opened at once, or read and write operations to other channels while the directory is open. DOS 2.0S or DOS 2.5 can get very confused if more than a single channel is open for reading the directory.

8.9 Wild Carts in File Names

Up to now, file names always specified files precisely, i.e. the full name of the file was given. FMS++ supports, however, a mechanism that allows you to identify a file by only parts of its name — for example if you had forgotten the precise file name, or want to include several files in a file operation.

FMS++ offers for this end three special characters that can be part of a file name: The question mark "`?`", the asterisk "`*`" and the dash "`-`". If the question mark is part of a file name, the file name identifies any file whose name is identical to the given name with the question mark standing for a single arbitrary character. For example,

```
LOAD "D:SH?P.BAS"
```

would load either `SHIP.BAS` or `SHOP.BAS` from disk, whatever file is found first. Or it would also match `SHXP.BAS`, if such a file would be present. More than one question mark may be present, and question marks may also be present at the end of base name and extender, where they also match missing characters:

```
LOAD "D:TEST??.BAS"
```

matches both `TEST.BAS` and `TESTED.BAS` because the question marks also match the missing two characters in `TEST`. It would not match `TESTING.BAS` because the `G` character is an additional character beyond the two wild card positions given by the question marks.

If this match is also desired, the asterisk "`*`" can be used: It matches any sequence of characters, up to the end of the base name or the extender. Thus, for example:

```
LOAD "D:T*.BAS"
```

would load the first file on disk whose file name starts with `T`, and whose extender is `BAS`. Last but not least,

```
LOAD "D:*.*"
```

would load the *first file* found on disk — and may create an error if this is not a BASIC program. Since the combination "`*.*`" matching any file is required so often, for example for showing the full directory contents, in the example of the previous section, it can be abbreviated to "`-`", which also matches all files. Going back to the directory program, the first line of which could be replaced by the more compact

```
10 OPEN #1,6,0,"D:-"
```

which is otherwise exactly identical.

8.10 Advanced Information on Wild Cards

The dash is a FMS++ extension that comes very handy in many situations as it allows quicker typing. One may wonder whether a wild cart always finds only the *first* file, or *any* file. The answer is that it depends on the operation: When channels to files are opened via the `OPEN` instruction, then FMS++ will scan for the first file in the directory that matches the wild card. For extended FMS operations like protection (file *locking*) or file renaming, wild cards match *any* file found in the directory. The same rule applies for the open modes 6 or 7 that open a channel to the disk directory. Reading from such a stream will return *any* matching file, not

just the first. The disk headline and the free sectors information will there always match the wild-card, and will be displayed anyhow. The headline can be suppressed by using mode 7 instead of 6.

This behavior — matching any instead of just the first file — can be changed by using the file name modifiers described in section 8.11. Especially the FMS++ rename command can, when used without care, generate multiple identically named files. In DOS 2.0S or DOS 2.5, you were simply lost, as you could only access the first of the two files, rename them back both, or delete them both, but you could not fix up the disk by only renaming the second. The file name modifiers of FMS++ resolve this situation nicely.

8.11 File Name Modifiers

Modifiers are a unique feature of FMS++ requesting additional features by extending the file name. A *modifier* consists of a letter or digit appended at the end of the file name after a slash "/". For example,

```
SAVE "D:TEST.BAS/V"
```

extends the file name `TEST.BAS` with the modifier `V`. This specific modifier turns on write with verify, i.e. the disk drive will check each written sector for correctness, and thus enhance the reliability of the disk I/O. This, however, also slows down the disk transfer and is thus disabled by default.

The following additional modifiers are also available:

Table 2: File Name Modifiers

Modifier	Function
V	Enable write verify
O	Disable write verify
A	Append to file, do not overwrite
D	Open the directory, not a file
N	Do not run binary file, load only
1	Apply only to first match
2	Apply only to second match
⋮	...
9	Apply only to ninth match

The `A` modifier has a double purpose and also disables listing the headline if applied to an open for directory, i.e. open mode 6. That is, it effectively changes mode 6 to mode 7. Open modes are explained in section 8.8. If `/A` is combined with the BASIC `LIST` instruction, untokenized BASIC files can be appended on the disk without loading them into RAM. The following sequence

```
LOAD "D:PROG1.BAS"
LIST "D:PROG.LST"
NEW
LOAD "D:PROG2.BAS"
LIST "D:PROG.LST/A"
```

appends `PROG2.BAS` at the end of the already saved program `PROG.LST`. When loading the combined file with

```
ENTER "D:PROG.LST"
```

lines from `PROG2.BAS` will extend `PROG1.BAS` by adding them to the program, and whenever a line number is used in both programs, the line from `PROG2.BAS` will take priority and replace the one from the earlier program. The same trick unfortunately does not work for tokenized programs, i.e. programs saved by `SAVE` instead of `LIST`. It does work, however, for *binary load* programs that can be saved from the DOS command line (see section 11) or the DOS menu (in section 12).

Modifier D turns a regular open into reading the disk directory instead of the file contents. This makes not much sense for BASIC, but it is very useful in combination with the DOS command line to print the directory on a printer. More on this in section 11.

Modifier N applies only to the *binary load* command of FMS++, explained in section 8.12. The details will be explained there. The modifier disables running a program then.

The modifiers 1 through 9 change the function of the wild cards introduced in section 8.9. While an OPEN without any modifier matches always the first matching file name in the disk directory, the modifier "/2" would instead match the second match. The line

```
LOAD "D:-/4"
```

would, for example, load the fourth file on disk, no matter what its name is. If this file is not a tokenized BASIC file, an error will result. The wild card modifiers are also useful when resolving duplicate file names. This feature will also be explained in section 8.12.

More than one modifier can be appended to a file name as well. If so, each modifier has to be introduced by a slash. Thus,

```
SAVE "D:-/O/2"
```

would overwrite the second file on disk with the BASIC program in memory, with write verify disabled. The order of the modifiers does usually *not* matter, unless the modifiers are in conflict, e.g. when specifying both O and V. In such a case, the latter modifier takes effect.

8.12 Additional FMS++ Commands

Besides the ordinary BASIC commands for manipulating files, like OPEN or PRINT #, FMS++ specific commands can be forwarded to the disk handler by means of the XIO instruction. This instruction takes five parameters:

```
XIO cmd,#iocb,aux1,aux2,"filename"
```

The first argument specifies the command the FMS is supposed to execute. Most of these commands are closely related to the commands available at the DOS command line (see section 11) and allow to issue modifications on the disk without entering DOS, albeit in a less convenient way. If you have DiskIO installed, see section 9, then additional BASIC instructions of similar function will become available.

For example, XIO 33 deletes files from disk and XIO 32 renames one or multiple files. The full list of available FMS++ functions is listed in table 8.12. The second argument identifies the I/O channel that shall be used for issuing the command. For most functions, the I/O channel does not matter, except that it must not be open for any other function or file; if the channel is already open, the XIO command will either fail, or not apply the desired function because the operating system then runs the command through the already opened channel, and not to the file name specified as last argument. Sometimes the function differs, depending on whether the channel is already open or not.

The arguments aux1 and aux2 should remain zero for most values of cmd. They typically have no further meaning, though exceptions exist. If so, a detailed explanation will be given.

Finally, the file name argument defines the file or the target the operation shall apply to. As always, the file name must include the device name, e.g. D:, separated from the actual file name by a colon. File name modifiers are allowed.

The following table gives an overview over all XIO commands; they are discussed in more detail below:

Table 3: XIO Commands

XIO Command	Function
3	Identical to `OPEN`. `aux1` and `aux2` have exactly the same meaning as the two arguments of `OPEN`.
5-11	Not usable from BASIC. These commands are equivalents of `PRINT #` and `INPUT #` that, due to differing calling conventions, cannot be used from BASIC programs safely.
12	Identical to `CLOSE`.
13	Verify a file name, also used by `STATUS #`.
32	Rename file(s).
33	Delete file(s).
34	Resolve wild-cards.
35	Lock files(s).
36	Unlock file(s).
37	Identical to `POINT`, but in this shape as `XIO` not usable from BASIC due to differences in the calling convention. Use `POINT` instead.
38	Identical to `NOTE`, but not usable in this form. Use `NOTE` directly. `POINT` and `NOTE` are introduced in section 8.16.
39	Initialize a disk, quick-format.
40	Resolve wild-cards, identical to `XIO 34`.
41	Load and optionally run a binary file.
42	Format a disk in single or enhanced density.
43	Format a disk in enhanced density.
254	Identical to `XIO 43`, format in enhanced density.

The `XIO` commands 5,7,9 and 11 are unfortunately not usable in this form, but they perform functions similar to `PRINT #` and `INPUT #`. `XIO 3` and `XIO 12` are exact work-alikes of `OPEN` and `CLOSE`, though the clear-text form should be preferred.

The `XIO` commands 35 and 36 lock or unlock files on disk. A *locked* file cannot be overwritten, or deleted, and any attempt to do so triggers an error. Locked files are displayed with an asterisk "`*`" in front of their file name in the directory listing. `XIO 36` removes the lock again. Note that this locking mechanism is purely software-based and does not prevent the disk from being formatted or initialized by `XIO` commands 39,42,43 or 254. If you want to protect an entire disk, it is better and more secure to cover the write-protection notch of the disk.

For example,

```
XIO 35,#1,0,0,"D:TEST.BAS"
```

protects the file `TEST.BAS` from accidental overwriting, and

```
XIO 35,#1,0,0,"D:-"
```

protects *all* files on disk because the wild card character − matches any file on disk. Note that modifiers (see section 8.11) may change the meaning of wild cards, thus

```
XIO 36,#1,0,0,"D:-/3"
```

will remove the write-protection lock from the third file on disk only.

`XIO 33` deletes a file, or several files, depending on whether wild cards or file name modifiers are used.

```
XIO 33,#1,0,0,"D:TEST.BAS"
```

will attempt to remove the file TEST.BAS from disk. If it either does not exist, or is write protected, an error will result. Wild cards may be used to delete multiple (or all) files.

XIO 32 renames a file, and takes two instead of one file name, separated by a comma, the initial file name, and then the new final file name. Thus,

```
XIO 32,#1,0,0,"D:TEST.BAS,FINAL.BAS"
```

will rename TEST.BAS to FINAL.BAS. Special care must be taken when combining this command with wild cards as this may result in multiple, identically named files. The following is a call for disaster:

```
XIO 32,#1,0,0,"D:-,WHATA.MES"
```

as it renames *all* files on disk to the same name, WHATA.MES. Should this mischief ever happen to you, a plain simple XIO 32 would not help since it, again, would apply to all files, and just change one mess into another. In Dos 2.0S and other Atari DOSes, you would be in a dead end. However, hope is near in the form of file name modifiers: You can pick one out of the many identically named files, and rename it back:

```
XIO 32,#1,0,0,"D:WHATA.MES/2,RESTORED.BAS"
```

will only pick out the second of all the similarly named files, and change its file name to RESTORED.BAS. Section 8.11 explains the function of file name modifiers in detail.

XIO 13 should, as all other XIOs except number 12 be only applied to closed channels, as otherwise it performs a different function not explained here, but see section 8.14. XIO 13 tests whether a file name is valid, the file is existing, and whether the file is write-protected. The following line will perform a test on the contents of the string variable A$:

```
XIO 13,#1,0,0,A$
```

If the contents of A$ is ill-formed, i.e. would include control characters or lower caps characters, the error code 165 is generated. If no disk is in the drive, error 144 results. Error code 138 is generated if the drive is turned off, error 170 if the file does not exist. Finally, error 167 results if the file is write protected, but no harm is done to the file itself in either case.

XIO 37 and XIO 38 are not usable in this form, but are available through the BASIC POINT and NOTE instructions. Both instructions are for advanced users only and position the file pointer within a file. The commands are explained in more detail in section 8.16.

XIO 34 and XIO 40 are identical and exist on two positions for backwards compatibility reasons. XIO 34 is the legacy position compatible to DOS 2.XL of the same author, XIO 40 supplies a similar, but not identical function under DOS 3 by Atari. This command checks its argument, which is typically a wild-card, and replaces the argument by the full file name the wild card refers to. Usage from BASIC requires some care to avoid overwriting program code. You need to supply a string variable large enough to capture the full resolved file name, and fill the initial part by the wild card you want to resolve. The following program segment demonstrates this:

```
10 DIM A$(20)
20 A$="D:-"
30 A$(4)=" ":A$(20)=" ":A$(5)=A$(4)
40 XIO 40,#1,0,5,A$
50 PRINT A$
```

This program will find the name of the fifth file on disk. The first line dimensions a new string variable of 20 characters, the next initializes it to the wild card that matches any file. Line 30 is a nice trick to initialize all the remaining parts of the string to blank spaces. This is necessary to make room for the output of the

XIO command. Line 40 resolves the wild card by means of XIO 40. The fourth argument to XIO, here the number five, identifies which of the matching files shall be found. If this argument is set to zero, then the matching file is defined by the file name modifier, which is an alternative way of picking out the right file. Line 50 prints the result.

XIO 41 loads a binary file from disk. The format of such files is defined in section 14; these are assembled programs in machine language that run natively on the system without requiring interpretation of the BASIC compiler. The third argument of the XIO command specifies which of the vectors in the binary load file shall be called when loading the program. This argument should usually be left at the value 192, indicating that both the program initialization and run vectors should be called. If set to 64, then only the run vector but not the init vector is called, for 128 the reverse happens. Finally, if set to zero, the program is only brought to memory and neither vectors are used. Binary files are an advanced feature, and more details are explained in section 14.

The following XIO command will, to give an example, load the menu-driven utility package from disk, provided the system disk is inserted in the drive:

```
XIO 41,#1,192,0,"D:MENU"
```

XIO 39 performs a quick format on an already formatted disk and erases all files quickly by erasing the disk administration information. It optionally installs a headline. Take care not to erase important data as this command will *not* check for protected files. It does check for the disk write protection notch, though. The following command will clear an already formatted disk in drive one, and will give it the name My Disk:

```
XIO 39,#1,0,0,"D:My Disk"
```

Note that lower caps characters, as well as all other characters except inverse video are allowed in disk headlines. The headline will appear in inverted in the directory listing. There is *no* XIO command to change the headline later on, the DOS command SETHDL could be used for this purpose instead.

While the previous XIO command could not prepare a fresh disk for usage as it requires an already formatted disk, the following three commands blank a disk thoroughly and imply a test for bad sectors, too. An error will result if the disk could not be formatted. As for the above command, if the disk was already formatted, all information on it will be erased and is lost forever. The first one, XIO 42, is the most versatile as it allows single or enhanced density format. The first format provides only 707 free sectors per disk, whereas the latter formats a disk to a format only compatible to the 1050 disk station, and then offers 963 free sectors on a disk:

```
XIO 42,#1,33,0,"D:My Disk"
```

performs a single-density format, and sets the disk name to My Disk. A disk is formatted in enhanced density instead of single density by

```
XIO 42,#1,34,0,"D:My Disk"
```

i.e. substituting the 33 by a 34. No other values for the second argument are accepted and may produce undesirable effects.

The XIO commands 43 and 254 are identical and only format in enhanced density. They do not care about their third and fourth argument at all:

```
XIO 254,#1,0,0,"D:My Disk"
```

performs an enhanced density format, and sets the disk name, which may also be missing — then no headline will be installed. XIO 43 should probably not be used and the legacy XIO 254 is preferable as it formatted disks under DOS 2.0S, too.

8.13 Advanced Information on XIO Commands

`XIO 13` is identical to the BASIC `STATUS` instruction, though its operation differs from whether the channel is open or not. If the channel is not open, the function mentioned in section 8.12 is run: A file is checked for existence and its protection status is returned. This is also the same operation `XIO 13` and `STATUS` performed on DOS 2.0S and DOS 2.5. Unfortunately, as `STATUS` does not supply a file name argument, it was basically non-functional on disks, and — unlike documented — did not return any useful information.

FMS++, however, implements as a work-around a differing function *if* the channel is already open: It then just returns the result code of the last I/O operation on the same channel. This allows `STATUS` to be used to check for the DOS *EOF warning*: `STATUS #1,A` will result in `A` set to 3 if the next read operation would run into the end of file, or will return 1 if the last operation succeeded. This makes `STATUS` a useful instruction again.

`XIO` commands 5 and 7 are the record and block oriented input operations of which only the former is available as `INPUT #1,A$`. Because `XIO` does not pass any length of the block to be transferred, the command is unusable in this form. The commands 9 and 11 are the corresponding write commands not available in BASIC at all. `PRINT #` works different and less efficient, i.e. is slower than what the Os could offer.

`XIO`s 37 and 38 are used by `POINT` and `NOTE` for random access, though the `XIO` command does not allow to fill in the arguments required by the corresponding disk commands, and thus both `XIO` variants are unusable. Just use `POINT` and `NOTE` directly, no advantage could be gained by `XIO` here or by talking to the operating system directly.

`XIO` 41 is the binary load command; many third party DOSes *incorrectly* populate XIO 40 with this command, however Atari reserved slot 41 for this as can be seen in DOS 3 which predates all these developments. DOS 2.XL and FMS++ follow this tradition initiated by Atari.

`XIO 40` was, under DOS 2.XL, the *Write DOS* command and injected the DOS bootstrap code into sectors 1 to 3, and the DOS code itself into the high-sector area not visible to files. Since FMS++ is not disk based, this XIO command became free and is now populated by `XIO 34` which performs an activity that is similar, but not identical to what DOS 3 had as an internal-only `XIO` code at this position.

`XIO 34` is a DOS 2.XL extension that is also used internally by the Dos command line to implement the copy command on multiple files. DOS 3 had also an internal command at this position which was officially never documented.

`XIO 42` is also a DOS 2.XL extension that works similar to a DOS 3 format extension, though the argument convention is different and specifies under DOS 3 the number of free blocks that become available after formatting. For DOS 2.XL and FMS++, it is the SIO format command issued to the disk drive, which is then later on used to derive the disk organization.

`XIO 43` is actually place holder for future extensions and simplifies the internal command handling. Avoid its usage, and use code 254 directly, which does right the same anyhow.

8.14 Additional BASIC instructions to work on files

In addition to `PRINT #1` and `INPUT #`, BASIC includes three additional elementary instructions to operate on channels — not even necessarily channels connected to the FMS: These are `GET`, `PUT` and `STATUS`. The first of them, `GET`, reads a single character from a channel and returns its ATASCII code. If this channel is connected to the keyboard handler, `K`, then this is command waits for the user to press any key, and the ATASCII code of the key is returned:

```
10 OPEN #1,4,0,"K:"
20 GET #1,A
30 PRINT "YOU PRESSED ";CHR$(A)
```

The above is a very simple program that demonstrates its use. The same also works with files, where data can be read sequentially from the start of the file to the end, character by character. If the file is a BASIC program written by the LIST command, the printed data will be the program itself:

```
10 OPEN #1,4,0,"D:TEST.LST"
20 GET #1,A
30 PRINT CHR$(A);
40 GOTO 20
```

If the program is a binary file, or a tokenized BASIC program generated by SAVE, the result will not be human-readable.

If you run the above program, you will notice that it will, after reading the file contents, abort with an ERROR 136. This is the indication that the end of file has been reached. This error can be captured by the BASIC TRAP instruction, but there is a more elegant way for testing whether the next read, let it be by GET or INPUT #, will hit the EOF:

```
10 OPEN #1,4,0,"D:TEST.LST"
20 STATUS #1,A
30 IF A=3 THEN 70
40 GET #1,B
50 PRINT CHR$(B);
60 GOTO 20
70 CLOSE #1
```

The STATUS command checks the return code of the last disk operation. On success, its argument is set to 1. If the last command returned an error, then error code will appear in the argument. The value 3, i.e. the value that is tested for in the above program, is generated when the next read operation will generate an error because we are right *in front of* the end of file.

While the GET instruction will read data from disk, the PUT instruction will do just the reverse and write data back to disk. Its argument specifies the ATASCII code of the next character to be written:

```
10 OPEN #1,8,0,"D:HELLO.TXT"
20 PUT #1,72
30 PUT #1,73
40 PUT #1,155
50 CLOSE #1
```

prints the text "HI" into the file, and appends an end-of-line character. This is because 72 is the code of H, 73 the code of I and 155 the end of line character. Of course, PRINT #1,"HI" would have done just the same, only simpler. Nevertheless, there are applications where the PUT command is useful, for example when manipulating binary files.

8.15 Information for Advanced Users

In the above program, STATUS was used to return the error or status code of the last FMS operation. Even though similar examples are listed in the Atari manuals of DOS 2 or DOS 3 variants, STATUS never worked as indicated above, and FMS++ fixes this. The reason for this problem is that STATUS is equivalent to XIO 13, which served another purpose under DOS 2.0S, DOS 2.5 and DOS 3: It checks whether its file name argument is valid, an existing file and not write protected, see section 8.12. However, as STATUS does not take a file name argument, there was no way of communicating a file to the FMS, and the instruction was useless.

Under FMS++, the function of XIO 13 or STATUS depends on whether the channel is open or not. If it is not open, then the XIO 13,#1,0,0,"D:FILE" form works and tests the file specification as indicated. If it is open, it works in the above sense by testing for the last I/O operation.

The EOF warning, i.e. the error code 3, is not unique to FMS++. What is unique, however, is that it also works consistently for the directory modes of OPEN, namely modes 6 and 7. This program

```
10 OPEN #1,6,0,"D:-"
20 STATUS #1,A
30 IF A=3 THEN 70
40 GET #1,B
50 PRINT CHR$(B);
60 GOTO 20
70 CLOSE #1
```

will list the directory contents — though slowly — but without requiring a TRAP command to react on the end of the directory list.

8.16 Random Access of Files

All the instructions seen so far either manipulate files, or modify them sequentially: Data is added to the file in the order writing or printing proceeds, from start to end. The next pair of instructions can record a location within a file, and can return to this location later on. A good use case is that of a data base: One file, the *index file* would contain all the locations of the records in the data base. The second file, the *data file* would contain the actual data, and the entries in the *index file* would refer to this file.

A pair of instructions exist to realize such applications: NOTE takes a channel and returns the current position within the file for later reference. For this end, it takes two arguments that store the position:

```
NOTE #1,A,B
```

The two variables can then be used to return to this very location later on, and operate on the contents again:

```
POINT #1,A,B
```

The values of A and B should be considered *opaque*, i.e. do not assume that they have any specific meaning, or that you can perform arithmetic on them and by that advance to *some other* location than the one computed by NOTE. Performing a POINT to a location not returned by NOTE before is most likely a call for disaster.

The following toy-example demonstrate how to use POINT and NOTE, arguably not in a very useful sense: This program reads the first line of text from a file, prints it, and then restores the read position back to the start of the file to read the very same line again. You need to press BREAK to abort the code:

```
10 DIM A$(40)
20 OPEN #1,4,0,"D:HELLO.TXT"
30 NOTE #1,A,B:REM TAKE NOTE ON THE CURRENT POSITION
40 INPUT #1,A$
50 PRINT A$
60 PRINT "AND AGAIN..."
70 POINT #1,A,B:REM RETURN TO THE SAME POSITION WE STARTED WITH
80 GOTO 40
```

Using POINT inappropriately returns in best case an ERROR 166, a point error, or an ERROR 164, bad file link. In worst case, it damages the disk structure when you try to write to a disk position that does not belong to the file at hand.

8.17 Advanced Information on POINT and NOTE

Actually, the arguments to POINT and NOTE are, as a matter of fact, the sector and byte offset of the file pointer. However, they should really *not* be treated as such because the FMS does not need to place files in contiguous sectors on disk. That is, if sector A belongs to a file, the file need not to continue at sector A+1. The sector of the file could also be in A-1 or completely elsewhere.

However, POINT and NOTE have also been extended somewhat. The OPEN mode 8, open for writing, also supports POINT and, naturally, NOTE. By means of POINT, the file pointer can be placed back to a position where data has already been written, and data need not to be appended only at the end of the file. FMS++ then automatically switches from appending to the file to overwriting mode. If the file pointer is placed back at the end of the file and data gets written, this data will then be automatically appended. Mode 13 also automatically switches from overwriting to appending when reaching the end of file. Neither DOS 2.0S nor DOS 2.5 allow the same amount of flexibility.

The DOS command line, and the DUP menu both make use of the advanced capabilities to copy files in several goes, and extending to a partially copied file. The DUP does this by first NOTEing the end of file before closing it, and then pointing back to the end position later on when extending the file.

8.18 The Direct Mode

While POINT and NOTE allow to extend the sequential structure of the files *somewhat*, they still have the huge drawback of not allowing arithmetic operations on their arguments. Without taking NOTE on the file positions when creating a file, one cannot computationally jump to a position within a file. Arithmetic operations on POINT arguments are not supported.

This drawback goes away in *Direct Mode*, which is a unique feature of FMS++. However, with great power comes great responsibility: In Direct Mode, the entire disk is yours, and the FMS does not help you administrating its contents. In specific, if you operate on a regular FMS++ formatted disk with the direct mode, you can see the otherwise internal administration information, and — in worst case — overwrite or damage it. Direct mode does not care about files, but rather the entire disk is *one single file*, without any further structure. The disk is accessed *directly*.

> *Data turned easily Directly into Trash* Do *not* use the direct mode on the same disks where you keep your programs. If you want to use the direct mode, reserve an *entire disk* for it, and use *only* this disk for this purpose. Writing with direct mode on a regular disk is calling for disaster. You will likely overwrite the FMS++ administration information, or data belonging to files on such a disk.

This clearly makes the direct mode an advanced feature, and it is recommended only for advanced users that know how to use it.

8.19 The OPEN Instruction in Direct Mode

The OPEN for direct mode is distinguished from the regular OPEN by setting the third argument to 128 instead of 0 as usual. Because the entire disk is then just one single file, the file name itself *does not matter, but has to be present*. In the examples here, it will always be "−", though any other valid file name could be used.

```
OPEN #1,4,128,"D:-"
```

opens the disk itself for reading. At this point, the file pointer on the disk is *nowhere*, and a POINT instruction must be used to position it at some byte or sector offset. For example,

```
SECTOR=360:BYTE=0
POINT #1,SECTOR,BYTE
```

would place the file pointer to the location where a regular FMS++ formatted disk would administrate the available sectors, also called the *VTOC*, short for *Volume Table of Contents*. The instruction

```
GET #1,A
```

returns the first byte of the VTOC, which is the VTOC version number. This byte is 2 on a valid FMS++ disk.

While opening the disk in direct mode for *reading* is harmless, the following instruction opens the disk for *writing*

```
OPEN #1,8,128,"D:-"
```

Besides the read-only and write-only mode, the mode 12 allows concurrent reading and writing. The following table shows all possible open modes in the direct mode:

Table 4: Direct Open Modes

Mode parameter	Purpose
4	Open for reading only. The disk will be opened for read operations only. Any write operation triggers an error. The file name is ignored.
8	Open the disk for write only. Since the disk is affected as a whole, any files on it are in danger. Use a separate disk for experiments.
12	Open the disk for reading and writing. This is the combination of the modes above.

No other modes exist. Modes 6 and 7 do not make sense since there are no files, hence no disk directory, only a single sequential disk to read from, and modes 9 and 13 do not make sense since the end of the disk is a physical parameter, and it cannot be made larger or smaller just by means of software.

8.20 POINT and NOTE in Direct Mode

Once the disk is open for reading or writing, `POINT` and `NOTE` can be used to work freely on it. Quite unlike the sequential mode, the arguments to `POINT` and `NOTE` follow here a very regular pattern: The second argument increments from zero to 127, then wraps around and, like an odometer, the first argument is then increment. While the second argument starts from zero, the first starts at one, and then increments up to the end of the disk. The *absolute* position on the disk can thus be computed by

```
NOTE #1,A,B
POSITION=(A-1)*128+B
```

and the absolute position can be transformed into `POINT` arguments by

```
A=INT(POSITION/128)+1
B=POSITION-128*(A-1)
POINT #1,A,B
```

9 DiskIO

DiskIO is a language extension for Atari BASIC originally developed by Bernhard Oppenheim for the ANTIC magazine. The release you find here as part of Os++ has been extended and made compatible with Os++ and FMS++. In specific, unlike its predecessors, DiskIO will now also operate under Mac/65, will stay resident when the Dos command line or DUP are invoked, and will use minimal memory when it finds the FMS++ overlay manager active (see section 13).

9.1 What is DiskIO

DiskIO is a BASIC and Mac/65 extension that adds eleven useful instructions to the language to ease the handling of files on disk. Unlike regular BASIC commands, DiskIO commands only operate in direct mode, i.e. they only work if you enter them without a line number. You cannot include them in BASIC or assembly programs. The additional instructions provide similar functionality as the commands of the DOS command line, and simplify disk handling within BASIC or Mac/65. Operating with the disk requires no longer any cryptic XIO commands, and listing the disk directory does not require you to go to DOS. You can also operate conveniently on the directory listing itself without having to type full file names.

9.2 How to Install DiskIO

DiskIO exists in the form of an `AUTORUN.SYS` file on the Os++ System Disk. If you insert the system disk when booting the system, and either BASIC or Mac/65 are present, DiskIO will be installed automatically. You can also copy DiskIO to your disks by the DOS command line or the DiskIO `COPY` instruction. DiskIO also exists as `DISKIO.EXE` on the very same system disk, for your convenience, should you ever want to load it manually because you have deleted the `AUTORUN.SYS` file.

For loading it manually, first go into the DOS Command line, insert the Os++ system disk in the first drive and simply enter there `DISKIO.EXE`, then press `RETURN`. Following that, enter `CAR` and press return, or simply press the `RESET` button to return to BASIC or Mac/65. DiskIO is now active.

9.3 DiskIO Instructions

DiskIO instructions operate a little bit different from BASIC or Mac/65 instructions in the way how they process arguments. With BASIC, you have to enclose the file name in double quotes, i.e.

```
SAVE "D:TEST.BAS"
```

whereas with Mac/65, you have to add a hash-mark `#` in front of the file name:

```
SAVE #D:TEST.ASM
```

With DiskIO, neither of the two is necessary. The file name can be written directly behind the instruction, without the `D:` and without the quotes or hash mark. Thus, the equivalent DiskIO instruction of the above would simply read

```
SAVE TEST.BAS
```

and implies the quotes and the hash-mark. However, you may still use the original form if you prefer, or want to save to any other but the first disk station.

The DiskIO instructions `LOAD` and `ENTER` are also exact equivalents of their native form and do not require quotes, hash-marks or a `D:` to operate on the disk. Just add the file name.

The DiskIO `LISTD` instruction prints the currently loaded BASIC program to disk, in human-readable form, very much like its BASIC cousin:

```
LISTD PROGRAM.LST
```

is equivalent to

```
LIST "D:PROGRAM.LST"
```

The reason for not calling it `LIST` directly is simply that then `LIST 100` might specify one of two things: First, list line 100, and second, save the current program under the name "`100`" to disk. Thus, `LISTD` replaces only the former function of the regular BASIC `LIST` instruction, and not the latter.

Similarly, the RUND instruction loads a BASIC file from disk, and then starts it. It is equivalent to RUN "D:...". The same reasoning as above made it necessary to call it RUND instead of RUN. Clearly, RUND is not available under Mac/65 since assembly programs cannot be started but require assembly first.

While the above instructions provided mainly convenience features, the next set of instructions is new and add functionality to BASIC: The LOCK command protects files from being deleted or overwritten. It works exactly as the LOCK command of the DOS command line. Hence

```
LOCK MYWORK.LST
```

protects the file "D:MYWORK.LST". Atari BASIC also supports this operation, but in a very inconvenient way. Instead, you would have to write

```
XIO 35,#1,0,0,"D:MYWORK.LST"
```

which is both longer to type and harder to remember. Unfortunately, only the latter form is acceptable as part of the BASIC program, LOCK works in the direct mode only.

The opposite of LOCK is UNLOCK, it removes the protection again. The DELETE command deletes files on disk. The RENAME instruction changes the name of a file, requiring both the old and the new name, separated by comma. Thus,

```
RENAME MYWORK.LST,MYWORK.DAT
```

changes the file extender of "MYWORK.LST" to "DAT". As above, neither a D: nor quotes are required, and both DELETE and RENAME, as well as LOCK or UNLOCK work from BASIC and Mac/65.

The BLOAD instruction loads a *binary* file from disk, as if its file name would have been typed into the DOS command line. The file could be a game, or any other type of program assembled or compiled to machine language. Be warned, however, that binary programs tend to overwrite the BASIC memory. It is advisable that you save your work first.

The COPY command finally copies entire files between disks, or within the same disk. For that, specify first the source file name, and then the location of the target, optionally including a device specifier:

```
COPY MYWORK.DAT,D2:
```

First, DiskIO will read the source file, then request you to insert the destination disk by printing

```
Insert disk 2, press any key(S=to E:).
```

To continue, press RETURN and the file will be saved to the second drive. If you press BREAK instead, the copy operation will be aborted and nothing will be written.

You can also copy files from one disk to itself, or between disks if you only own a single drive. For example:

```
COPY MYWORK.DAT,FINAL.DAT
```

will copy the file to another file, named FINAL.DAT. When DiskIO requests you to press RETURN, you may either leave the source disk in the drive or insert just another disk. In the first case, FINAL.DAT will appear on the same disk as the original file. In the latter case, it will be written to the second disk.

Finally, if the second argument is omitted, the target file will be named identical to the source file. This makes of course only sense if you swap disks after reading the source.

The COPY instruction includes one specialty, namely it allows you to write the files to the screen instead of to disk. This is useful for actually peeking into the files — provided they contain human-readable data. For that, press the S key instead of RETURN when DiskIO requests you to insert the target. For example,

```
COPY PROGRAM.LST
```

would usually copy "PROGRAM.LST" from one disk to another. However, when DiskIO prints its line

```
Insert disk 2, press any key(S=to E:).
```

and you press now the S key on the keyboard, the contents of PROGRAM.LST will appear on the screen for inspection. The file will actually not be copied. The very same can be achieved, of course, by

```
COPY PROGRAM.LST,E:
```

directly.

The DIR instruction will list the directory of the disk to the screen. If there are more files on disk than fit on the screen, you'll be prompted

```
Press ESC -> Abort  RETURN -> Continue
```

to either press the RETURN key to continue the listing, or ESC to abort it. The output of the DIR instruction will be alphabetically sorted, and you might notice that each file name is preceded by a number. These numbers allow quicker handling of files by "dot" instructions that will be discussed in section 9.5. Like all other DiskIO instructions, they only work in direct mode. You cannot, unfortunately, add a DIR instruction to your BASIC program.

Last but not least, DiskIO extends the BASIC and Mac/65 LIST commands by allowing an empty second argument, requesting them to list the entered program from the line number given as first argument up to the end of the program:

```
LIST 200,
```

will list the program from line 200 on up to the end of the file. Both BASIC and Mac/65 required two arguments and did not allow the above form.

9.4 Abbreviated DiskIO Instruction

Very much like BASIC instructions, DiskIO instructions can be abbreviated by a shorter name and a dot "." behind it. However, while BASIC is fairly permissive and flexible about how instructions can be abbreviated, DiskIO only accepts a single abbreviation for each instruction. As in BASIC, abbreviated instructions must be terminated by a dot. Table 9.4 lists the accepted DiskIO abbreviations:

Table 5: Abbreviated DiskIO Instructions

Long Instruction	Abbreviation	BASIC Equivalent
LISTD	LI.	LIST "D:..."
SAVE	S.	SAVE "D:..."
LOAD	L.	LOAD "D:..."
ENTER	E.	ENTER "D:..."
RUND	*none*	RUN "D:..."
LOCK	*none*	XIO 35,#5,0,0,"D:..."
UNLOCK	UN.	XIO 36,#5,0,0,"D:..."
RENAME	RE.	XIO 32,#5,0,0,"D:..."
DELETE	DE.	XIO 33,#5,0,0,"D:..."
BLOAD	BL.	XIO 41,#5,192,0,"D:.."
COPY	CO.	*none*

As for all DiskIO instructions, neither do the abbreviated forms work within the BASIC program. They are only recognized in the direct command mode.

9.5 DiskIO Dot Commands

Typing in file names is tedious and error-prone, even more so if the `DIR` instruction already printed them onto the screen. To speed up your work, DiskIO can also address directly files from the directory listing on the screen. These "dot" instructions are even shorter versions of the DiskIO instructions introduced above, but only work as long as the DiskIO directory listing is visible on the screen.

Here is an example how to use them: Consider you type `DIR` in BASIC and get the directory listing, which might read as follows:

```
            System Disk
 1 AUTORUN SYS 034  2 HANDLERSSYS 001
 3 TEST    BAS 002 926 FREE SECTORS
```

This disk has apparently three files on them, among them the BASIC program `TEST.BAS`. Now, you could either type `LOAD TEST.BAS` to read it from disk and load it into memory, but even shorter, you could just enter

```
.L3
```

which would do the same. `.L` is the "dot" instruction for `LOAD`, and the number `3` is the file number, i.e. its index, in the directory listing printed above. Note that there is no space (blank) accepted between `L` and the file number; this and all other dot instructions only work as long as the directory listing is visible on the screen. Naturally, they also only work in direct mode and cannot be used in programs.

All other DiskIO instructions also have "dot" instructions that operate directly on the directory listing, see table 9.5 for the full list. The `RENAME` and `COPY` dot instructions are in so far special as they take two arguments, not one, and only the first argument is a file index, the second is a full file name. For example, with the above directory listing on the screen,

```
.C3,BACKUP.BAS
```

will copy file number three, i.e. your BASIC program, into a new file named `BACKUP.BAS`. As you might have guessed, `.C` is the dot instruction for `COPY`.

Table 6: DiskIO Dot Instructions

Long Instruction	Dot Instruction
LISTD	.X
SAVE	.S
LOAD	.L
ENTER	.E
RUND	.R
LOCK	.K
UNLOCK	.U
RENAME	.N
DELETE	.D
BLOAD	.B
COPY	.C

Before executing potentially dangerous instructions, DiskIO will ask you for permission just to make sure you have picked the right file to work with. So for example, again using the above directory listing on screen, `.D3` will make DiskIO to ask for reconfirmation first:

```
DELETE"D:TEST.BAS?
```

DiskIO waits now for an input from you - anything but the Y key will abort the operation. For example, just hit BREAK or N to stop DiskIO from deleting your precious program.

DiskIO will ask for permission before executing the .S (SAVE), .B (BLOAD), .C (COPY), .N (RENAME) or .X (LISTD) instructions, too, because they can either damage the integrity of your program — as BLOAD might — or may overwrite existing files on disk. All other dot instructions are executed immediately.

9.6 Asking for Help

One final instruction remains, namely the HELP instruction. This instruction will print a brief command reference on the screen. Pressing the HELP key followed by the RETURN key also works. Due to technical reasons, the HELP key alone is, unfortunately, not sufficient.

9.7 Leaving to DOS and returning to BASIC

If you enter the DOS command line or DUP Menu, DiskIO will become inactive but stays resident. It will not interfere with DOS or the DUP menu in any way. As soon as you return to BASIC, DiskIO will be operational again.

10 Overview on the DOS User Interfaces

Os++ comes not only with a file management system that handles the organization of individual files on disk, and which was described in section 8, but also with user interfaces besides BASIC that make file handling simpler. This part is traditionally called the *Disk Utility Package*, or *DUP* shortly. These packages allow you to list the disk directory, rename, delete or copy files, run binary programs or format disks. In fact, Os++ comes with *two* such interfaces: A command line driven interface that addresses the needs of more experienced users, and a menu based interface that is more appropriate for beginners. Each interface has its advantages and drawbacks.

The command line interface is ROM resident and does not take any RAM space; it is always present, and does not erase your BASIC program unless you allow it to. Thus, it is quick and easy to move from BASIC to the command processor and back, no disk operation is required for that, and your program will remain unharmed. However, the learning curve for the command line interface is steeper.

The menu driven user interface is more user friendly, as all options are available in a convenient menu on the screen that lists all available operations. However, the menu based interface is disk-based and thus must be loaded from disk every time you need it, very much like the DUP of DOS 2.0S or DOS 2.5. Unlike the command like interface, it will erase your program, and you should save your program to disk before running the menu as your code will be lost otherwise. Note that the menu does not offer a MEM.SAV file as DOS 2.0S did. Loading the menu from disk also takes longer than running the ROM based command line interface, though customizing the menu allows you to cut the loading time down at the price of loosing some functionality. Also, some advanced tools are only available in the menu interface.

The next section will describe the command line interface, then followed by section 12 and following where the menu interface will be described.

11 The DOS Command Line

The DOS command line interface allows you to manipulate files on disk very much like *DiskIO*, see section 9 allowed you to handle files from BASIC. Unlike DiskIO, however, it is part of the ROM based operating system and does not need to be loaded from disk. It is always present, and does not take RAM space. You

reach it from BASIC by typing the command DOS, then press RETURN. The command line will then prompt you with a screen like the following:

```
Thor DOS 2.++ V 1.8 Enhanced Density
Copyright (c) 1990-2014 by THOR

D1:
```

and the cursor will be waiting for input behind the D1: prompt. At this time, you can enter a command and press return, just like you can enter commands at the BASIC READY prompt. However, DOS commands are different and work different from BASIC instructions, the logic of which is described in the next section. At this time, it is sufficient to know how to return to BASIC: You go back from the command line to BASIC by typing CAR followed by RETURN, or just by pressing the RESET key. Your BASIC program will remain unharmed.

If you had a cartridge inserted while turning on the computer, control will be handed over to the cartridge instead of BASIC. How to leave the cart to DOS depends on the cartridge type, but for most programming languages a command like DOS command will bring you to this command line interface. You can re-enter the cart by typing CAR on this command line.

If you disabled BASIC by pressing the OPTION key while turning on the machine and had no disk in the disk drive to boot from, the operating system will enter the DOS command line instead. From there, you can work with the files on the disk, or run programs or games stored on disk. The CAR command or RESET will simply return to the command line.

11.1 The Command Line Syntax

Once you entered the command line, the prompt D1: tells you the current drive all commands will operate with. This is by default the first disk drive connected to the machine. For example, the command

```
DIR
```

to be entered behind the D1: prompt will list the directory of the drive as shown by the prompt. This prompt, and thus the default device to operate on, can be changed — more on this later.

DOS commands take, in general, file names as arguments. While in BASIC a file name always requires a specification of the device the file is located on, e.g as in

```
SAVE "D:PROGRAM.BAS"
```

the DOS command line allows an abbreviated syntax: Double quotes *must not* be included, and device names as D: *need not* to be included. If a device name is missing, the command operates on the current drive as shown by the prompt.

For example, the command

```
DIR *.BAS
```

will list all BASIC programs on the disk as shown by the prompt, again D1: unless you changed it. The expression *.BAS is a *wild card* that specifies the DIR command which files to list. Wild cards had been discussed in section 8.9.

You can also include a device name within the argument of the command, e.g.

```
DIR D2:*.BAS
```

will list all BASIC programs on the second disk drive — if connected and turned on. As a side effect, the command prompt will change to `D2:` after the execution of the `DIR` command, and all further commands will automatically address the second disk drive, until you change it back.

You can also change the current device without running a command. For that, just type in the device name at the command prompt, followed by a colon.

```
D1:
```

to be entered behind the `D2:` prompt left over from the above command will change the default back to the first drive.

The DOS command line understands of course many more commands than just `DIR`, to be discussed in the following. Just remember that you can go back to BASIC any time by pressing `RESET` or by entering the command `CAR`, the second command to be remembered after `DIR`.

11.2 Internal and External Commands

The DOS command line has a couple of commands built into its program that can be executed directly. However, the command line is extensible by *external* commands which will be loaded from disk, specifically from the disk that is shown at the command prompt.

That is, if you type in the following at the `D1:` prompt in the DOS command line

```
FOO.EXE
```

then the DOS will try load a file named `FOO.EXE` from the first disk drive and hand over command execution to this file. If such a file is not present or loading of this file fails, an error will be printed.

As DOS commands are simply binary files, this mechanism allows you also to execute arbitrary programs from disk. Just type in their file name, and DOS will try to execute them as external commands. This process will include loading them from disk to RAM, and starting them. Hence, typing a file name executes the named program.

> *DOS is not a DOT-COM* Dos++ does *not* follow the convention of the Os A/+ command processors by appending a `.COM` appendix to all commands given at the command line. If you want to load or execute a file having the appendix `.COM`, you need to type in this appendix explicitly. Instead, Dos++ follows the Unix convention of commands *having no appendix* at all.

While all binary files can be loaded as commands from disk, they will typically not interpret the command line arguments given to them. Instead, they will just be run, and probably present a user interface of their own. Commands specifically suited for Dos++ are shown in the directory listing by *not having any* file appendix, i.e. the columns that show usually the letters `BAS` or `SYS` or `EXE` are blank for such commands. You can execute them simply by typing their name, without any appendix. You find a couple of external commands on the system disk — these will be discussed below.

> *Might Damage Your Program* Since Dos++ has no control on where an external command will be loaded to in memory, it is not too unlikely that your BASIC program will be overwritten by such a command. Up to the `MENU` command, all external Dos++ commands are safe, though, and will not damage your program at all. In case of doubt, save your work before entering the command line.

11.3 Advanced Information on Internal and External Commands

A word on external and internal commands: The Dos++ command line will first try to resolve a command *externally* by looking for a file as given at the command prompt, on the drive given by the drive name ahead. Only if that fails, Dos++ will search its internal list of commands. That is, if you type in

```
DIR D:*.*
```

two things will happen: First, the disk is searched for a file named `DIR`, and DOS will try to execute this file, passing over the command line argument `D:*.*` to it. Usually, there is no such file, and hence, as a second step, the internal database of commands is scanned, where finally the `DIR` command is found, and then executed. This would, for example, allow to extend Dos++ by an external, probably better or more powerful `DIR` command.

If you specifically want to run the *internal* command and want to disallow Dos++ to scan for external commands, add a *double quote* in front of the command to be executed. This suppresses external command resolution:

```
"DIR D:*.*
```

means “yes, I really mean the internal `DIR`, do not try to load one from disk”.

Last but not least, internal commands have only three sensitive letters. Whether you type in `DIR` or `DIRECTORY` does not matter; unless of course, there is an external file that matches this command.

11.4 Elementary Internal Commands

This section describes the list of commands built into the ROM space that are always available, and for which you do not need to insert the system disk to run them. Two of them you already know: `DIR` lists the directory of the current drive, or of the drive given as its argument, and `CAR` returns to the inserted cartridge, if there is one.

The `LOCK` command protects the file name or file names given as its arguments from getting deleted, renamed or overwritten. It works pretty much like the `LOCK` command of DiskIO.

```
LOCK D:*.BAS
```

protects all BASIC files on the current drive.

The opposite of `LOCK` is `UNLOCK` which removes this protection again. For example,

```
UNLOCK D:AUTORUN.SYS
```

removes the write-protection from the `AUTORUN.SYS` file on disk — if there is one. Better don’t do this, unless you know that you don’t want this file: `AUTORUN.SYS` on the system disk contains the DiskIO program.

The `DELETE` command will remove files and make the disk space they occupy available for other files. Similar to the above, it also takes a single file name or a wild card as argument. Be warned, however, `DELETE` does not ask for permission but proceeds immediately.

```
DELETE D:JUNK.TMP
```

removes some temporary junk file you might have left on the disk.

The `RENAME` command changes the name of a file. It expects two arguments, the old name, and the new one, separated by comma.

```
RENAME TEST.BAS,FINAL.BAS
```

renames your BASIC test program into `FINAL.BAS`, assuming that you're done writing it.

> *Don't mix Rename and Wild Cards* If the source of the rename command is a wild card, and the destination is not, it may happen that you generate multiple identically named files on disk. These can be resolved as normal files using the file name modifiers explained in section 8.11, but this is usually a source of confusion. Avoid to mix rename with wild card arguments — this is asking for trouble.

11.5 Copying Files with the Dos Command Line

The `COPY` command copies one or multiple files on a disk, or from one disk to another. It takes one or two arguments. The first argument is the file or the files to be copied — wild cards are allowed here and if more than a single file matches, all matching files will be copied. If the destination is missing, `COPY` assumes you are copying the files from one disk to another, but use the same disk drive and want to swap disks. That is,

```
COPY *.BAS
```

will copy all BASIC programs from the current disk to another, and once a file has been read into memory, the Dos++ command line will ask you to insert the destination disk and confirm by pressing return:

```
Insert Target ,press RETURN
```

That is, remove the source disk, insert the destination disk into the same drive, then press `RETURN` and the copy process will continue by writing the files back. Once done, Dos++ will ask you to swap back the source disk to continue reading the next file:

```
Insert Source ,press RETURN
```

and so forth, until all files are copied.

If you have two disk drives, you can copy between them without having to swap disks. Just give the target drive as second argument to `COPY`:

```
COPY D:*.BAS,D2:
```

will copy all BASIC programs from the first to the second drive. There is no need here to swap disks at all.

You can also copy one file on another on the same disk. For that, specify a target file name as follows:

```
COPY TEST.BAS,BACKUP.BAS
```

will make a backup from your program `TEST.BAS`. Since Dos++ does not know whether the target disk you want to write the backup to is the same disk or another, it will ask you to swap disks after reading the source:

```
Insert Target ,press RETURN
```

If you just leave the same disk in the drive, it will simply copy the file there.

Finally, a word about copying huge files: If the file does not fit into memory completely, Dos++ will require multiple rounds to copy it, and will ask you to swap the disks back and forth, until the entire file has been copied. For very huge files, three rounds are not unusual, but the number of times you have to swap source and destination depend on the memory available to copy.

> *Copy is Safe, but Slow* The `COPY` command is program friendly and will not overwrite your program. It will just reserve memory on top of your program area. The drawback is that it may require you to swap disks more often than necessary: At least once for every file to be copied, and probably more than once if a file does not fit continuously into memory. The Dos++ Menu discussed in section 12 has a smarter copy command that minimizes the amount of disk swaps required.

A word of warning: Due to an unfortunate bug in Mac/65, the `COPY` command will partially overwrite the assembler sources. This is because Mac/65 does not indicate the memory it requires to store your sources correctly. DiskIO works around this bug, but the Dos command line can not since it is not aware which cartridge it is currently running from. If you want to run `COPY` with a Mac/65 source in memory, it is advisable to first save the souces to disk, then go to DOS, enter `NEW` to clear them from RAM and then perform the copy.

11.6 Dangerous Internal Commands

The next two internal commands wipe out entire disks and should be handled with care: `FORMAT` formats a new disk and writes the disk management structures on it so it is ready for use. You can optionally provide a device name, i.e. the device name the blank disk is inserted in. For example,

```
FORMAT D2:
```

formats a disk in drive 2. You may also, optionally, provide a *headline* that will be shown on top of the directory each time you list the directory contents. The headline goes behind the device name:

```
FORMAT D2:MyPrograms
```

or directly behind the `FORMAT` command if you want to format the disk in the current drive:

```
FORMAT MyPrograms
```

The `CLEAR` command works like a quick-format that erases all contents from the disk by re-initializing all administrative information, but does not attempt to physically low-level format it. Otherwise, it works identically to `FORMAT` and also allows you to define a headline:

```
CLEAR NewDisk
```

> *No Spaces Please* Currently, neither `FORMAT` nor `CLEAR` accept blank spaces in the headline. However, you can use the external command `SETHDL` to change the headline later. This program *does* accept blank spaces. Another possibility is to just substitute spaces by an underscore "_" character.

Please remember that `FORMAT` and `CLEAR` erase disk contents, without waiting for any further confirmation.

11.7 Internal Commands for Working with Binary Files

The next three commands are for experts only, and manipulate "binary load" files that either represent external commands, compiled or assembled programs or games in general. Less experienced users might want to skip this subsection.

The `LOAD` command loads a binary file into memory, and optionally runs it. This command is actually superfluous since you can also load a file by typing its name directly on the command prompt, but it is here for completeness. That is, whether you type in

```
GAME.EXE
```

or

```
LOAD GAME.EXE
```

makes no difference whatsoever. Both perform the same.

As introduced in section 8.11, you may always specify the `N` modifier to suppress starting a binary load file. That is,

```
LOAD GAME.EXE/N
```

will load the file to memory, but will not start it. Its init vector, if any, will be called, however. You can start this program then later on with the RUN command.

The opposite of LOAD is SAVE. This command saves a portion of the current memory to disk, such that it can be restored later on. Optionally, a run address can be given at which the program shall be started later on. An init address can be included as well, which is a machine code program that is executed first before attempting to run the main program. This init address could display a title or prepare the machine for the actual main program. Details on how binary load files work are given in section 14.

The SAVE command takes three mandatory and two optional arguments. The mandatory arguments are the file name, and the start and end address, inclusively, of the memory to be saved to disk. All addresses are to be given as four hexadecimal digits:

```
SAVE PAGE6.EXE,0600,06FF
```

This command saves the contents of the famous "page 6" to disk, ready to be loaded the next time you need it. Probably a tiny machine language program you wrote?

Optionally, a run address can be given as fourth argument. The program will be started from this address when loaded from memory.

```
SAVE PAGE6.EXE,0600,06FF,0680
```

This will include the run address $680. Hopefully, some executable code is at this address as otherwise the machine will crash when loading the file.

If five instead of four arguments are given, the fourth argument is the init address and the last one is the run address. That is,

```
SAVE PAGE6.EXE,0600,06FF,0600,0680
```

saves page 6 to disk, and instructs the Dos the next time it loads this file to run initialization code at address $600, followed by starting the program at $680. Please note that the init address comes first, then the run address.

Finally, the RUN command re-runs previously loaded machine code programs, or an arbitrary program in memory whose address you remember. If you call it without arguments,

```
RUN
```

it just re-runs the previously loaded binary program. For example, if you used the N modifier to suppress running it, the RUN command will pick up the run address left in memory and launch it now. If no program has been loaded before, RUN without arguments does nothing.

If you provide an argument, this is the address you want to run from, which has to be given as four hexadecimal digits. For example,

```
RUN E474
```

starts execution at the reset vector of the operating system as if you pressed the RESET key. Otherwise, you should better know what you start or you can crash the machine easily.

11.8 Miscellaneous Internal Commands

Two internal commands are still missing to complete the catalog: The CAR command you already know, and the NEW command. The former just returns to the cartridge, for example BASIC or Mac/65. If no cartridge is inserted, the screen will be cleared and the system will return to the Dos command line.

The system disk provides the external command BASIC that will turn on the built-in BASIC interpreter of the XL series even if option has been held down during bootstrap.

The NEW command works similar to the BASIC instruction of the same name: It wipes out the memory and removes your program from it. The purpose of the NEW command is to make some room for the the COPY command, i.e. to allow it to use your program space for copying data. It is also useful to wipe out memory and instruct the BASIC interpreter to start from a blank program if you loaded a machine code program from disk that might have overwritten your program space. Sometimes the BASIC interpreter is then left in a total state of confusion that can only be resolved by NEW. The NEW command works, by the way, cartridge independent. It can also be used to wipe out an assembly language program edited with Mac/65 from memory.

11.9 Hints and Tricks for the Command Line

You might have noticed that the DIR command has no means to write its output to another file, or to print it. However, printing the directory or listing it to a file can be accomplished by a neat "trick": The D modifier, also introduced in section 8.11 redirects the input from a regular file to the directory, and hence can be used to COPY the directory contents to a file:

```
COPY D:-/D,DIR.DAT
```

takes the directory, and creates an ATASCII file named DIR.DAT that contains the directory. Note that - is the wild card that matches any file, and that D means that instead of copying any file, the COPY command should only copy the directory entries that match the wild card.

The above becomes more useful if another device is used as destination, for example the printer:

```
COPY D:-/D,P:
```

copies the directory contents to the P: device, which is the printer.

A second modifier comes handy if you want append data at the end of a file. Consider for example that you have two large BASIC listings PROG1.LST and PROG2.LST and you want to append the second table at the end of the first to merge the two programs. The A modifier changes the regular "overwrite" output mode to the "append" mode and makes it possible to use the COPY command again to append data instead of overwriting it:

```
COPY PROG2.LST,PROG1.LST/A
```

does just the above: Instead of erasing the destination first, COPY will now append the data at the end of the first program.

A small note of warning: Merging BASIC programs with COPY like the above only works for files created with the LIST command, generally identified by the appendix .LST. It does not work the way you expect by regular .BAS files. However, binary load files can be merged exactly like this.

Last but not least, a hint in case you performed the mischief of renaming one file to a file already present: You are now in the situation of having two identically named files on disk. Any type of operation now either only addresses the first file, as for example LOAD, or both files at once, as for example DELETE or RENAME. Thus, it seems, you cannot get rid of one without also the other.

Again, modifiers help: The 1 modifier explicitly only matches the first file, the 2 the second. That is, you could give the second file another name and by that resolve the ambiguity:

```
RENAME TWINS.DAT/2,NEWNAME.DAT
```

gives the second twin a new name, hopefully resolving the trouble. Alternatively, you could also rename the other twin

```
RENAME TWINS.DAT/1,EVILTWIN.DAT
```

or delete it right away:

```
DELETE TWINS.DAT/1
```

The `1` says FMS++ to only work on the first of the two identically named files.

11.10 External Commands

Unlike the built-in internal commands, external commands must be loaded from disk and come in the form of binary, i.e. machine language programs. This section describes the external commands provided on the Dos++ system disk. Because these commands are, as said, on disk and not in ROM, the system disk must be inserted to make them work. Since the DOS command line always looks for external commands at the current drive, make sure that the command prompt showing the current drive names the system disk. In the examples given in this section, the system disk is assumed to be in drive 1, and the command prompt should show `D1:`. To change it, just enter the drive name followed by a colon, and press `RETURN`. See section 11.1 for an introduction into the DOS command line syntax.

Unlike internal commands, external commands must be loaded into the system RAM and therefore may conflict with BASIC or assembler programs stored there. The external commands provided with the system disk are, however, written with great care to avoid such damage, with the only exception of the `MENU` command that runs the menu driven interface. All other commands will leave your programs unharmed. The menu based interface is introduced in section 12 and more details on it are found there. All other external commands are described here one after another.

The `HELP` command lists a brief overview on all `internal` commands built into the command line. It does not take any parameters.

The `REBOOT` command reboots the system immediately, without asking back. It works as if turning the machine off and on again. Lacking the system disk, you can trigger a reboot with the internal `RUN` as well, though in a less convenient way.

```
RUN E477
```

will do right the same as `REBOOT`.

The `BASIC` command returns to the BASIC interpreter or the inserted cart. Unlike the `CAR` command, it is able to activate BASIC later on even if it has been disabled during bootstrap by holding down the `OPTION` key. It thus allows to enable the BASIC ROM after bootstrap.

The `SETHDL` command is able to change the headline of a disk without requiring a complete initialization. While the internal `FORMAT` and `CLEAR` commands always erase all files on the disk, `SETHDL` can change a headline afterwards, and may also remove a headline. Installing a headline on a disk that was formatted without one requires a bit more care, but is possible as well. Unlike `FORMAT` or `CLEAR`, `SETHDL` also accepts blank spaces in the headline.

It can be called with or without a parameter. If no parameter is given, `SETHDL` will prompt you to enter a drive and a new headline:

```
Enter (device:)headline or press
RETURN to clear :
```

You can now take the chance to exchange the system disk by the disk whose headline you want to change, and provide the command with the new title to appear topmost in directory listings of this disk:

```
Enter (device:)headline or press
RETURN to clear : New title
```

will change the title of the current drive to `New title`. If you want to change the title of a disk inserted in another drive, just add the drive name in front of the title:

```
Enter (device:)headline or press
RETURN to clear : D2:New title
```

This will change the headline of the disk in drive 2. Unless you had your system disk inserted in drive 2, you won't need to exchange disks in this case.

To remove the headline, just press `RETURN` without entering anything, or just provide the device name alone. To remove the disk headline from the disk in drive 2, just enter

```
Enter (device:)headline or press
RETURN to clear : D2:
```

The `SETHDL` command can also be run with the device name and the new headline on the command line, and then performs the operation immediately without prompting you to enter anything. That is,

```
SETHDL D2:New Title
```

changes the head line of the disk in drive 2 right away without asking further for any input. If you have only one drive, this form does not allow you to change the headline of any disk except the system disk because you won't have a chance to swap disks before the command executes.

In case you want to add a headline to a disk that was formatted without one, the `SETHDL` command will likely fail with an error such as

```
Headline position is used!
```

which means that a regular file currently blocks the entry in the disk directory that is reserved for the headline under FMS++. The following trick allows you to insert a headline nevertheless:

First locate the first (topmost) file on the disk by listing the directory with the `DIR` command. The output might read as follows:

```
* MYPROG.BAS 010
953 FREE SECTORS
```

Copy this file to the same disk under a second unique name. We will change the name back later, please do not worry. Thus, enter in this example:

```
COPY MYPROG.BAS,NEW.BAS
```

and wait for the copy to proceed. Then, potentially unlock the old file, and delete it under its old name:

```
UNLOCK MYPROG.BAS
DELETE MYPROG.BAS
```

This step released the directory position for the headline. Now change the name of the copied file back to its original name:

```
RENAME NEW.BAS,MYPROG.BAS
```

Finally, insert the system disk to load the `SETHDL` command:

```
SETHDL
```

Now change disks back again, insert the original disk back again, and enter a new title. This time, `SETHDL` should succeed.

11.11 Advanced Information: Accessing Program Parameters

When calling either internal or external commands, additional parameters may or must be included to specify the activity of the command, the files the command shall operate on, and much more. On the command line, one or multiple spaces separate the command from its arguments.

Due to constraint ROM space, and unfortunately quite unlike Os/A+, Dos++ does not include a fancy interface to parse off command line parameters. It is up to external commands, thus binary load files, to read them from RAM and parse them appropriately. The full command line is located at the following RAM location:

```
$0580   DUPCMDLINE
```

The command line buffer contains as first part the default device to be used if any other device specification is missing, separated by a colon from the command name itself. Command arguments follow behind the spaces, if any. The end of the command line is indicated by an EOL character `$9B`.

The syntax of the command line arguments, and its parsing is up to the command itself. Dos++ does not enforce a specific syntax, but it is recommended to reflect the syntax of the internal Dos commands, namely to separate arguments by comma, and include modifiers that change the operation of the program by slashes.

12 The Menu Interface of Dos++

The menu driven interface to Dos++ is a more convenient and more accessible way to manipulate files on disks. Unlike the command line interface, the menu based interface is disk based and requires additional RAM space. Unlike most other external commands, see section 11.10, it *will* erase BASIC memory to run, and you should better save your work before starting the menu. Dos++ has no provisions to save the RAM content itself, i.e. the `MEM.SAV` file DOS 2.0S and DOS 2.5 offered to restore BASIC memory when returning from the menu is *absent*.

To start the menu, first go into the Dos command line by typing

```
DOS
```

The Dos command line will appear. At this time, since the menu is disk based, insert the system disk in the first disk drive. Now type

```
MENU
```

and the Dos++ menu will load. After a while, the full menu has appeared, and should look by default as follows:

```
THOR-Dup Version 2.++ (c) 1990,2013 THOR
A.Directory         B.To Cartridge
C.Run DOS           D.Delete File(s)
E.Rename File(s)    F.Lock File(s)
G.Unlokc File(s)    H.Format Single
I.Format Disk       J.Clear Disk
K.Binary Save       L.Binary Load
M.Run at address    N.Load OS/A++ file
O.Duplicate File(s) P.Duplicate Disk
Q.Check Drive       R.Radix Convert
S.Drive Diagnostics T.Set Disk Headline
U.Check Density     V.Set Default Device
W.Configure DOS     X.System Information
```

Each menu item stands for an operation Dos can run on a file or a disk. The letter upfront tells you which key to type to activate the corresponding operation. For example, to view the contents of a disk and list its directory, hit the key A on the keyboard. Dos will then prompt you:

```
directory -- search spec.,list file.
```

which asks for further parameters which files to list, and where to print the directory listing to. Details on the directory operation will be discussed in the next section. For now, it is enough to remember to press RETURN *twice* to list the contents of the disk in drive #1.

The disk directory will then appear on the screen, in two columns, with the further instruction

```
Select item or RETURN to clear
```

The menu itself remains invisible at this point to let you read the directory, but you can press the SELECT console key to toggle the menu on and off at any time. The directory listing will remain unaffected by the menu, and turning the menu off again will make the full directory content re-appear again. You may, at this point, select any other disk operation, or you may just press RETURN to clear the screen and make the menu re-appear. Unlike SELECT, which also displays the menu, the RETURN key will additionally clear out the screen contents except for the menu above.

To return to the Dos command line, type the key C. Dos will then ask you for confirmation. If you answer N, the menu interface will remain active, if you press Y, the menu will quit and you will find yourself back in the command line interface. To return from there back to the inserted cartridge — or BASIC if was enabled — enter CAR.

A quicker way to return to BASIC right away is the menu item B. As with Run DOS, it will ask for confirmation and will return you right away to BASIC or the inserted cartridge. Remember, however, that unlike with the command line interface, your program *is lost* if you haven't saved it before starting the menu.

12.1 Advanced Information on the Menu

Unlike the DUP of many other Dos implementations, the Dos++ menu is fully customizable and you can either re-organize the menu, remove menu items, or "pull out" menu items exclusively. The latter operation helps to save memory, especially for the file and disk copy menu items. If you load these menu items exclusively, more memory will be available for them, and less disk swaps are necessary to copy files or disks.

The menu is, for this end, organized in "item groups" each of which enables one, or multiple menu items in the full menu. Each of the item groups is represented by a file with the appendix MEN on the system disk. For example, the file DIR.MEN on the system disk contains the menu items A through C, and the file DUPLICAT.MEN the file copier. These are *not* ordinary binary load files, but use a custom structure. One additional file, namely MENULST.SYS, tells the Dos how to compose the menu. Editing this file will customize the menu, may include additional menu items, may change their order, or may remove menu items to speed up loading. Customization of the Dos menu is discussed in section 12.12.

To "pull out" an item group, press the / forward-slash key on the keyboard, which you reach by SHIFT and 7. Dos will then prompt you to enter the name of the item group you want to load exclusively:

```
Which menu to load :
```

Enter now the name of the file containing the item group. As said earlier, they all end with .MEN by convention. So for example, load the system disk in drive 1 and enter

```
Which menu to load :DIR.MEN
```

and press RETURN. Dos will now remove all loaded items from its menu, and will present only the items of the loaded item group. It will also re-enumerate them from the letter A on. Thus, for the above example, the menu will be shortened to just the three entries present in DIR.MEN:

```
THOR-Dup Version 2.++ (c) 1990,2013 THOR
A.Directory          B.To Cartridge
C.Run DOS
```

The prompt will change, too, to remind you that you have currently pulled out a menu:

```
Select item or BREAK to return
```

While the SELECT and RETURN key work as before to toggle the menu on and off and to clear the screen, the BREAK key serves a new purpose: It aborts the execution of a pulled out item group and returns to the full menu. For that, make sure that you insert the system disk first where Dos can load the items from.

As already mentioned, the main purpose of pulling out commands is to save some memory for memory intensive operations, like copying files or disks. The following sections will describe the Dos menu items in the order of their appearance in the menu, and will group them according to their item group. The name of the file containing the item group will also be given.

12.2 Elementary Operations: Directory, and Quitting the Menu

The menu items in the directory group provide, by default, the menu items A through C. They enable you to list the directory contents or quit the menu. They will be described here one by one:

```
A.Directory
```

enables you to list the directory content, on the screen, or print it on a printer, or write the list back into a file on disk. If you hit the A key, the menu will respond by

```
directory -- search spec.,list file.
```

The following options exist from here on: You can either press a key 1 through 4 to list the contents of the disk in the corresponding drive to screen, or you can press RETURN to select from further options. In the latter case, the cursor will appear right below in the next line. At this point, you may enter a wild card that specifies which files to list, optionally with a device name upfront, and an optional target file to output the directory listing to.

For just listing files on disk, specify a wild card as follows:

```
directory -- search spec.,list file.
*.MEN
```

and press RETURN. This will list all the files on the disk that have the appendix .MEN, and will print them to screen. These files represent the menu items.

To list the directory to a printer, add the device name of the printer, separated by a comma:

```
directory -- search spec.,list file.
*.MEN,P:
```

Of course, instead of specifying the printer, any other output device or file would also work:

```
directory -- search spec.,list file.
*.MEN,D:DIR.LST
```

would write the directory into the file DIR.LST on disk itself.

Finally, you can also list the directory of any other drive by including its device name in the wild card:

```
directory -- search spec.,list file.
D2:-
```

will list the contents of the disk in drive two. Remember that the wild card − matches any file. As a short cut, this specific wild card can also be omitted, and the above is equivalent to

```
directory -- search spec.,list file.
D2:
```

Two other items will be discussed here: Leaving the menu to a cartridge or BASIC, and leaving the menu to the command line. For the former, press B. Dos will respond by asking for reconfirmation:

```
O.K. to run cartridge ?
```

Type the Y key on the keyboard to proceed. Any other key aborts the operation and returns to the menu.

For the latter, press C. Again, Dos will ask for confirmation:

```
O.K. to run DOS ?
```

The Y key will bring you back to the Dos command line, and any other key will return you to the menu.

Advanced users are reminded that the directory related menu items are contained in the file DIR.MEN on the system disk.

12.3 Simple File Operations: Deleting, Renaming, Locking and Unlocking

The menu items D through G provide elementary file manipulation operations, namely to delete or rename files. Locking and unlocking files provide a mechanism to protect files from getting overwritten, renamed or erased by accident. Locked files appear with an asterisk * upfront in the directory listing, and any attempt to alter them — except unlocking — returns an error. File locking is an elementary software-based protection of files on disk. However, locking does not prevent the Dos from formatting the disk the files are located on, or getting erased or damaged by physical means.

The menu item D erases one or multiple files from disk. When selected, the Dos prompts you with

```
Which file to delete ?
```

and there waits for you to enter a file name or a wild card. You may optionally add a device name if the files are not to be deleted from the default device. For example,

```
Which file to delete ?
*.TMP
```

will erase all files ending on .TMP on the disk in the first drive. Unlike the command line and DiskIO, however, the menu will ask you for every file whether to proceed or not, and this will prevent you from deleting files by accident. That is, should the file FILE1.TMP match, it will respond with

```
Type "Y" to delete D1:FILE1.TMP
```

At this time, you can press the Y key to proceed, the N key to skip this file, or the BREAK key to abort the deletion operation altogether.

You may, additionally, include a device name to remove files from any other drive rather than the default one:

```
Which file to delete ?
D2:*.TMP
```

will remove files from the second rather than the default drive. Last but not least, you can also tell Dos that it should not ask you on every file, but should proceed to delete files directly without waiting for confirmation:

```
Which file to delete ?
*.TMP/Y
```

The /Y is an additional modifier, see also section 8.11, that suppresses the confirmation prompt. Its inverse, which is also the default for Delete is the /N modifier which requests prompting.

The E menu item changes names of one or multiple files. Unlike D, it requires two inputs, the old and the new name, but it also prompts for reconfirmation.

```
Rename - give old name, new
*.DAT,*.DTA
```

will rename all files with the appendix .DAT to files with an appendix .DTA, but will leave the rest of the file names unaltered. It will also prompt for each renaming operation.

```
Type "Y" to rename D1:FILE1.DAT
 to *.DTA
```

Again, you can answer with Y, N or abort the entire operation with BREAK. You can suppress the prompt and force renaming all files right away with the /Y modifier:

```
Rename - give old name, new
*.DAT/Y,*.DTA
```

As with all rename commands, renaming files may result in multiple files having the same name. In case you want to rename a single file out of multiple identically named files, use the /1 through /9 modifiers, as introduced in section 8.11.

```
Rename - give old name, new
*.DAT/2,*.DTA
```

This only renames the second file ending on .DAT.

Items F and G lock or unlock files, that is, they add or remove a software write protection very much like the Dos command line LOCK and UNLOCK commands do. If you press F, Dos will prompt you to supply a file name or a wild card matching all the files you want to lock from write access:

```
What file to lock ?
```

As above, the menu takes now either a file name, a wild card, with or without a device name. If no device name is given, the default device is assumed, which is the first disk drive. In the following example, all BASIC files are write protected:

```
What file to lock ?
*.BAS
```

Since locking or unlocking is a reversible operation and cannot potentially harm your data, Dos will not prompt you to confirm the operation. However, you can ask it by including the /N modifier. The following locks all files on the second disk drive and prompts you for each file individually:

```
What file to lock ?
D2:-/N
```

Unlock, i.e. the menu item G is the inverse of lock, and removes the write protection.

Advanced users that wish to customize the Dos menu find all the above items in the menu item group file XIO.MEN

12.4 Formatting Disks

The menu items H through J clear entire disks and are thus potentially dangerous. In specific, they do not check whether the disk contains any locked files. Only the write protection notch of disks can prevent these operations from proceeding. Formatting disks is required for fresh media you bought from the store and which do not yet have a valid disk structure written to them. Formatting creates the physical structure necessary to allow disk access. In addition to a complete format, Dos++ also offers a quick format that only re-initializes the administration information and thus quickly erases all files from disk, but does not perform a physical low-level format. Clearing a disk is a quick alternative for disks that have been already low-level formatted before.

Low-level formatting comes in two variants: Single or enhanced density formatting. Single density disks can be read by any Atari disk drive, they offer a capacity of roughly 86KB and format a disk to 720 sectors, 707 of then will be available for free usage, the rest is required for administration information. Enhanced density disks are only readable by the 1050 or newer drives. The physical format defines 1040 sectors per disk of which 963 remain available for use. This provides a storage capacity of approximately 117KB per disk.

Note that unlike other Days, Dos++ is ROM based and hence does not need to write a DOS.SYS file to disk, and hence no storage capacity is lost for the file management system itself.

The H menu item performs a single-density low level format, the I item an enhanced density format, if the drive supports it. Otherwise, they perform identically. In the following, let us see how the I item performs:

```
Format - enter (device:)headline
```

Dos asks here for an — optional — device name, and for an optional headline, a disk title that is printed on top of all directory listings. If you just press RETURN, the disk in the default drive will be formatted and no headline will be added. You would usually remove the system disk first, of course. If you changed your mind, pressing BREAK will abort the operation.

Having entered all required information, Dos asks you one last time to reconfirm the operation:

```
Press "Y" to format D1:
```

Again, you can abort by pressing N or BREAK. The operation starts with Y.

The following

```
Format - enter (device:)headline
D2:My Storage
```

formats a disk in drive two, and adds the headline My Storage. Spaces and lower caps characters are allowed here. If you don't need a headline, entering

```
Format - enter (device:)headline
D2:
```

will do right the same. Finally, the device name can also be omitted as well:

```
Format - enter (device:)headline
My Storage
```

This will format the disk in the default device, which is the first disk drive. Be alert not to format your system disk and remove it first.

Cleaning the disk works quite likely formatting it: After selecting the menu item, the Dos prompts you to enter a device name and a headline:

```
Clear disk - enter (device:)headline
```

The information required here is again quite the same, an optional name of the disk drive containing the disk to clear, and an optional headline to put on top of the directory listing. Please remember that clearing a disk requires that it has already been formatted before. It just quickly removes all files from the disk by clearing the administration information.

All the above menu items are contained in the file `FORMAT.MEN` in case you want to customize the menu. For more on this, see section 12.12.

12.5 Loading and Saving Binary Files

Binary files are machine language programs that can be loaded and run from disk. Such files contain games, programming languages or any other program written in the native language of the machine. The menu items `K` through `M` allow you to load, save or run such programs. Some advanced knowledge is surely helpful. The most useful of this item group is item `L` which loads a machine language program into main memory and starts it. Such binary programs have, by convention, the appendix `.EXE` or `.COM`. It first asks you on the name of file to load:

```
Which file to load ?
```

Dos expects here an optional device name, and the file name. For example,

```
Which file to load ?
GAME.EXE
```

will load and run `GAME.EXE` from the default device, i.e. `D:`. In case you only want to load the file to memory, but do not want to run it, use the modifier `/N` as described in section 8.11 to suppress starting the file:

```
Which file to load ?
D2:GAME.EXE/N
```

This will load the file `GAME.EXE` from the disk in the second drive, but will not start it. The loaded binary can be started later on by the menu item `M`.

A note of warning: Binary files may or may not require the same memory the menu itself requires. The Dos command line described in section 11 only requires minimal memory since it is ROM based, but the menu is not resident and disk based and hence may conflict with external files. While loading and starting an external file will usually work fine since the binary load algorithm is again in ROM, returning correctly to the Dos menu may not. In such cases it is advisable to run the binary menu items as an external menu item as described in section 12.1.

In case loading a binary file fails, try the following: First insert the system disk into the first drive, then press `SHIFT 7`, i.e. the forwards-slash. Dos asks you now to provide the name of an external menu item group. The name of the binary menu item group is `BINARY.MEN`, so provide this as input:

```
Which menu to load :BINARY.MEN
```

The menu will now change to list only the three items in the binary group:

```
THOR-Dup Version 2.++ (c) 1990,2013 THOR
A.Binary Save       B.Binary Load
C.Run at address
```

Press B now to start the binary load, and then proceed as above. More on external item groups is found in section 12.1. You can always return to the full menu by inserting the system disk and pressing BREAK.

The menu item M in the full menu starts a previously loaded binary, or executes a machine language program at a given starting address. This menu item requires advanced knowledge in machine programming, and it is easy to crash the machine by using it inappropriately. If you select M, the Dos menu asks you to provide an address to start execution from:

```
Run from which address ?
```

You can now either abort the operation by pressing BREAK, or provide a hexadecimal number, or just press return to run a previously loaded binary program. The following address will, for example, perform a cold start:

```
Run from which address ?
E477
```

The last operation in this item group is the reverse of loading a binary file, namely saving a block of memory on disk as a binary file, and provide optionally an init and a run vector. When loading the file later on, the init vector will be called first and should prepare the execution of the program. The run vector will be called last, unless the modifier /N is appended to the file name when loading the file. This modifier suppresses starting, but not initializing a program. While certainly an advanced feature, let us keep a brief look at it:

```
Binary save - give start,end(,init,run)
```

The Dos menu expects here at least three arguments, separated by comma: An optional device name, and the mandatory name of the file to be created as first argument; followed by the first address to include in the file as hexadecimal number, and the last address (inclusive) to be saved. If four arguments are given, the fourth argument is the run address in hexadecimal, for a total of five arguments the fourth argument is the init address, and the last the run address:

```
Binary save - give start,end(,init,run)
MYGAME.EXE,3F00,7FFF,3F00
```

This creates a binary executable program starting at address $3f00 and reaching up to and including $7fff. The contents of these memory cells will be restored to their state they have when creating the file. The program will be run at $3f00. Note that the program will *not* be run if you do not include a run address, i.e. unlike Os/A+, Dos++ will not automatically start programs from the first address. In the example above, no init address is given.

As for “Binary Load”, here a note of caution: The Dos menu will require some memory of its own and hence may overwrite any program you had in RAM before. It is therefore advisable to use the Dos command line instead to save a binary program as its memory usage is minimal. Loading the binary item group as external command helps little in this case as the memory will already be overwritten when the full menu loads the first time. A second note of caution: The Dos menu requires the popular page 6 — addresses $0600 through $06ff — for its internal vector table, and hence will overwrite any program that resides in this area. The Dos command line does not alter page 6, though some external commands may.

12.6 Executing Os/A+ Commands

DOS/A+, DOS XL and DOS XE are Dos products of Optimized System Software that use a somewhat different mechanism for executing binary files and passing arguments to them. The Dos menu provides a compatibility layer that allows running such files that is available as menu item N. Otherwise, the same restrictions as with the regular binary load apply, namely that Os++ commands may require the same memory

as the menu itself. It is thus advisable to execute the Os++ compatibility layer as an external item, see 12.5 or section 12.1. The name of the menu item file is `OSPLUS.MEN`.

When started, Dos first asks you to provide the name of the file to load, followed by any arguments you may provide. The arguments are here separated by a blank space from the command, and the complete file name is to be given. Unlike DOS/A+, Dos++ does not append a `.COM` to the file name. After pressing `N`, Dos now responds with the prompt:

```
Load from which OS/A+ file ?
```

At this time, you can not only provide a file name, but also the arguments to the OS/A+ command you want to execute. Please remember that the Dos menu *does not* append the `.COM` appendix as Os/A+ does:

```
Load from which OS/A+ file ?
COPY.COM D:PROG1.BAS D:PROG2.BAS
```

Provided you inserted the OS/A+ system disk in the default drive, this will call the OS/A+ command `COPY` and will supply two arguments to it, the source file name to copy, and the destination to copy to. `COPY` will then start with its work. Since the Dos command line has the `COPY` command built-in, this is only a demonstration of the possibilities and not a serious application of the menu item.

> *A Horse of a Different Color* Dos++ and OS/A+ interfaces are different, and while you can run binaries from either Dos on the other, passing *command arguments* between the Dos and its commands works different. Thus, the need for the above interface layer.

12.7 Duplicating Files

The menu item `O` copies files between disks or devices, or from a disk onto the very same disk under a different name. It works pretty much like the `COPY` command from the Dos command line, but is much smarter on RAM usage and caching of files. If you are copying multiple files, the Dos menu will try to fit as many into the available RAM as possible, minimizing the amount of disk swaps necessary. The Dos command line can only copy files one by one.

Once you type `O`, the Dos menu will ask you to provide the source files to be copied, and their destination:

```
Duplicate - give from(,to) :
```

The source can be a file name which may also contain wild cards, with an optional source device where the files are to be read from. If this is the only information specified, Dos will copy from the source device onto the very same device, possibly requiring you to swap disks. If you provide a destination, the file or files will be copied *between* disks.

```
Duplicate - give from(,to) :
*.BAS
```

will copy all files with the appendix `.BAS`, that is, by convention BASIC programs, from the default drive onto the very same drive, requiring you to swap disks. This is the right option if you have own only a single disk drive.

```
Duplicate - give from(,to) :
*.BAS,D2:
```

will copy all BASIC programs from the default drive onto the second drive. This will not require you to swap disks.

You can also provide a destination file name which makes sense if you copy *a single* file only. In this case, the same file will be copied between disks or onto the same disk, but under a different name:

```
Duplicate - give from(,to) :
GAME.EXE,GAME.COM
```

will create a duplicate of `GAME.EXE` and will give it a new name `GAME.COM`. You are given the chance to swap disks, and if you do, the copy operation will between disks. Otherwise, the file is copied onto the same disk but under a different name.

Finally, you may also include a device identifier in the destination which will then copy between devices, and hence between disks, changing the name:

```
Duplicate - give from(,to) :
D:GAME.EXE,D2:GAME.COM
```

will copy the file `GAME.EXE` from the first disk drive to the second, and will change its name on the way.

For demonstration purposes, let us continue with the very first example of copying BASIC programs between disks, so enter `*.BAS` as response, but *do not yet* press `RETURN`. First, insert the source disk to copy the files from, and *then* proceed with `RETURN`. Dos will first determine which files to copy, and will then ask you for each file, one after another, weather to include it in the copy process:

```
Duplicate - give from(,to) :
*.BAS

Type "Y" to copy D:MYPROG.BAS
to D:MYPROG.BAS
```

If you wish to copy the file, press `Y`, otherwise press `N` to skip this file and advance to the next. While `N` only skips files, pressing `BREAK` aborts the operation completely.

Dos will now proceed with all the files, and do not be afraid if you are not yet requested to insert a destination disk — Dos tries to use the memory optimally and reads in as many files as it can to avoid the tedious swapping. It will then read in the file:

```
Type "Y" to copy D:MYPROG.BAS
to D:MYPROG.BAS
Reading D:MYPROG.BAS
```

and will continue to ask you for each matching file it finds. If it runs either out of memory space, or finds no further match, Dos requests you to insert the destination disk:

```
Insert destination and press RETURN
```

You may, at this point, still abort the operation by `BREAK`, or swap disks and continue with `RETURN`. Then, Dos will write the files one after another and indicates on what is happening:

```
Writing D:MYPROG.BAS
```

plus potentially many more messages. If Dos could not fit all files into memory at once, it will ask you to re-insert the source disk to continue reading files:

```
Insert source and press RETURN
```

and the play will continue. Otherwise, the copy operation will come to an end.

If you want to copy all files anyhow, it might be tiresome to press `Y` over and over again. You can tell Dos in advance to *really* include every file, and stop asking you. For that, add the modifier `/Y` to the source specification:

```
Duplicate - give from(,to) :
D:*.EXE/Y,D2:
```

will copy all files ending on `.EXE` from the source disk to the destination, and will proceed without asking for confirmation.

The more memory available, the less you have to swap disks when copying files. It is thus a good idea to run the file copier as an external menu item and thus remove all the unneeded items from RAM, see section 12.1. The name of the item group that contains the file copier is `DUPLICAT.MEN`.

12.8 Duplicating Entire Disks

While the `O` menu item copies individual files between disks, the `P` menu item copies entire disks. In fact, it does not even require the Dos disk structure and may even copy disks that are not structured in the Dos++ layout. If you press `P`, Dos first asks you to provide the source and the target device to copy between. The destination drive may be identical to the source, in which case you would need to swap disks.

```
Copy disk - give from,to(,first,last)
```

Source and target devices need to be specified by their device letters, file names or wild cards cannot be included here. The two additional options `first` and `last` can restrict the amount of the data to be copied and require excellent knowledge of the disk structure. Only advanced users should specify them, and their discussion is for this reason delayed to the end of the section.

The following input will copy a disk in the first disk drive, onto a disk to be inserted into the very same drive. Thus, it will require you to swap disks:

```
Copy disk - give from,to(,first,last)
D:,D:
```

and will then ask you to insert the source disk

```
Insert source        and press RETURN
```

You may abort the operation here by pressing `BREAK`, but for now proceed to create a backup of the system disk. The next thing Dos asks

```
Is the source a Dos 2.++ disk? If so,
type "Y" to skip unused sectors:
```

If you are sure that the disk structure of the source conforms to Dos++, you may press `Y` here to speed up the copy process. In case you are not sure, it is always safe to say `N` here. The disk operation may then take longer, but will surely include all information on the disk. The Dos++ system disk surely conforms to its own standard, it is thus OK to say `Y` here. The next information Dos asks is

```
Clean unused or empty sectors? If so,
type "Y" to write them to disk :
```

If you use a freshly formatted disk, you can safely say `Y` to avoid blanking data is already cleaned by the disk format anyhow. If you copy Dos++ disks, it also safe to say `Y` here as administration information ensures that blank sectors are identified correctly in either case. If you are copying a non-Dos disk, and the target disk was used before, it is likely safer to say `N` here. In either case, `N` is *never a wrong choice*, it will at worst take a little bit longer to copy the disk.

After answering this last question, Dos++ will start reading data from the source disk and will display its progress in the top half of the screen. Dos will read as many sectors as fit into memory, and will then ask you to swap disks and insert the destination:

```
Insert destination and press RETURN
```

You may still abort the operation now by pressing BREAK. Otherwise, insert now the target disk and press RETURN. Dos will now start writing to the destination. After a while, you may be requested to insert the source disk again and confirm with RETURN that you swapped disks. Disk swapping might be necessary several times, depending on how much memory is available and depending how many blank sectors Dos finds it does not need to store.

To copy from the first to the second drive, specify D2: as destination:

```
Copy disk - give from,to(,first,last)
D:,D2:
```

The very next thing Dos requests from you now is to insert both disks, the source and the destination. Potentially, you may then read a message like

```
Destination format is single density
which does not match the source density.

Press "Y" to continue nevertheless :
```

This happens if the disk format of the source and the destination disk is not identical. In almost all cases this implies that you cannot safely copy the disk. Copying data from a single density onto an enhanced density disk is harmless, but leaves unused sectors Dos++ will not be able to use as the disk administration information is copied directly and will still indicate a single density disk. Thus, coping data by this menu item from single to enhanced disks will not provide any advantage. The reverse, copying from enhanced to a single density disk, is even more dangerous as some information cannot be fit onto the target, and hence, is lost. Thus, if you see the above message, it is in almost all cases *not* desirable to continue, and better to abort the operation. It is better to use the file copier of menu item O to copy the disk contents one by one, and by that also rebuild the administration information on the disk. Alternatively, you may also format the destination first by the menu items H for single and I for enhanced density, and then proceed with copying the disk.

Otherwise, copying between drives proceeds in the very same way as copying from one drive onto itself, except that Dos will not request you to swap disks.

If you have only a single drive available, it is usually a good idea to run the disk copier as an external menu item — by removing all unused items from memory, more RAM will become available to the disk copier and the number of disk swaps will be reduced. For more details on external menu items, see section 12.1. The name of the item group file containing the disk copier is DUPDISK.MEN.

Finally, a last word on the two additional parameters you may provide for copying disks: These specify the first and the last sector on the disk to be copied, to be given in hexadecimal. You should really know what you are doing if you attempt to copy a disk only partially as the administration information will then no longer fit to the data recorded on disk. Individual files should be copied by the file copier, menu item P, and not by this advanced feature. For example

```
Copy disk - give from,to(,first,last)
D:,D2:,1,3
```

will copy the first three sectors from the disk in the first drive to the disk in the second drive. These are traditionally the boot sectors. However, as Dos++ is ROM based, it does not require these sectors, which are therefore usually blank. Thus, actually, nothing useful is copied here.

12.9 Miscellaneous Utility Functions

The menu items Q through V implement various utility and management functions that help you to test the disk and drive integrity, convert numbers between the hexadecimal and decimal number system, and install or change the disk headline.

The Q menu item checks the disk for physical faults, bad and empty sectors. It does not perform an analysis of the higher level logical disk structure, though. After pressing Q, insert the disk you want to test into any drive, and type in the drive name, such as D1: to test the disk in the first drive. Operation will then start immediately.

The output will list an overview on the content of each of the 40 tracks of the disk, where the track number is printed to the right, and the number of the first sector of each track to the left. All numbers are in hexadecimal. The text between the sector number and the track number represent the contents of all sectors in the track. For example, the following line

```
153: **.....................*****  tr.:0D
```

denotes the the content of track $0D, that is, track 13 in decimal. This track starts, on this disk, with sector $153, which is 339 in decimal. Sectors that are completely empty are indicated by a dot ., asterisks * indicate full sectors. Sectors marked by a plus sign + are sectors that are zero *except* for administration information, i.e. belong to a file that contains zeros at this specific spot. Minus signs − indicate defective sectors the drive could not read. Please note that the output of the disk analysis depends of course on the disk you inserted and is likely different from the above example. Also note that this is an analysis on the physical layer of the disk, and the output does not reflect whether the sector is actually still part of a file. That is, if you delete a file from disk and hence release its sectors for further use, the disk status command will continue to show these sectors as non-empty, even though they are ready for recycling. The disk status will also list sectors that are required to administrate the disk structure, and hence will appear full even though they are not part of any file.

Once the disk has been read completely, the disk tester will provide a conclusive result, giving the total number of empty, used and defect sectors it found. Numbers are again in hexadecimal:

```
$0157 full  sectors,
$02B9 empty sectors and
$0000 error sectors.
```

Note again that the number of empty sectors here *does not* coincide with the number of free sectors the disk directory listing shows. The former prints the number of physical sectors which contain only zeros, the latter the number of disk sectors still available for storing file data.

While the hexadecimal output might be useful for some applications, it is — especially for beginners — somewhat inconvenient to read and understand. The R menu item allows you to convert between the hexadecimal and decimal number basis. For that, provide the number to convert as input, plain for decimal numbers, starting with a $ sign for hexadecimal numbers:

```
Give number, hex starts with "$" :
$157
Decimal 343 = Hex $0157
```

Here a hexadecimal number is converted to decimal, the output lists the provided number both in decimal and in hex. Without the $ sign, the input is assumed to be in the usual decimal format already:

```
Give number, hex starts with "$" :
1536
Decimal 1536 = Hex $0600
```

Again, output is given in both the decimal and the hexadecimal number basis.

The menu item S runs a hardware drive diagnostics test on the selected drive. This test works *only* with an original Atari 1050 drive, and will likely not work on third-party hardware, or at least will not return the expected result. This menu item runs undocumented Atari-internal test commands that are not implemented on any drive but the 1050. In specific, the diagnostics include testing the track zero sensor, which tells the

drive when the the read/write head has reached the out-most track of the disk, it tests the start and stop time of the motor, and the rotation rate of the spinning disk. To run all these tests, insert a disk into the drive to be tested — the disk will not be damaged or modified and is only required for the purpose of the test — and press the drive number to be checked. The output will look approximately as follows:

```
Raw motor speed-up time : 8980
Value value should be < 9000.

Raw rotation rate       : 2080
Value should be between 2025 and 2082.

Head settle test passed.

Track 0 sensor test passed.
```

The first test checks how long the drive needs to bring the spindle motor up to speed. This motor rotates the disk itself. If the value is too high, the motor takes too long to accelerate the disk. This may indicate a defective motor, or a defective voltage regulator.

The second value is an indication of the disk rotation speed once the motor is up to its final speed. Again, a value too low or to high indicates a problem with either the motor or the power supply in the disk.

The third test checks whether the drive read/write head can be positioned correctly on the disk tracks, and can find the tracks correctly. The last test checks whether the track zero sensor works correctly. This sensor indicates that the drive head has reached its out-most position. If this fails, then the sensor is probably misaligned or dirty.

The `T` menu item allows you to change the disk headline, to remove it or to add one. For that, Dos asks you to provide a drive name and the new headline:

```
Set headline - give (device:)headline
```

The device name should always be `D` optionally followed by a unit number between `1` and `8`, and a colon. If no device name is supplied, the default device is used. That is,

```
Set headline - give (device:)headline
My disk
```

will change the headline of the disk in the first drive to "`My disk`". If you supply no headline, it will be cleared out. For example,

```
Set headline - give (device:)headline
D2:
```

will clear the headline of the disk in drive two.

While unlike the `SETHDL` command, the Dos menu also accepts headlines including spaces, other restrictions still apply: If the first slot of the disk directory is already in use, installing a headline onto a disk without one will fail. The trick given in section 11.10 helps again: Copy the very first file on the disk onto the same disk under a new name, then delete the first file and rename the copied file back to the old name. For details, see section 11.10. Afterwards, the first directory slot will be free to take a headline.

The `U` menu item checks the physical layout of the disk. Dos++ supports single density and enhanced density disks, where the first leave 707 and the latter 963 sectors for the user. The physical disk layout, however, allow 720 and 1040 sectors, respectively. The remaining sectors are used for administration purposes or are left empty. This menu simply prints the format of the disk in the selected drive. For that, first inserted the disk to be tested in any drive, then type in the drive number to be tested. The operation will return immediately:

```
Enter drive # to check:1

Enhanced density,
1040 sectors,
26 sectors per track.
```

In this example, the disk in the first drive is an enhanced density drive. Single density drives have 720 sectors in total and 18 sectors per track.

Finally, the `V` item defines the default device to apply commands to if no further drive is given. To rename, copy, lock or unlock files, and for many other operations, the device name is optional. If no device name is given, Dos automatically inserts the default device name. When starting Dos, the default device is `D:`, i.e. the first disk drive. This menu item allows you to change this behavior and use any other drive as default device, for example to leave the Dos system disk in drive one, and use the second disk as default. For example, type `V`, then try the following:

```
Give default device (current D1:)
D2:
```

The default device has now changed to the second disk drive. Any menu function not requiring an explicit disk number will now use the disk in drive two instead. To try, type `A` to list the directory, and press `RETURN` leaving all options empty. If drive two is available and connected, the contents of the disk in this device are listed instead of the contents of the first one. The same also applies to all other commands, such as “rename”, “lock”, “unlock” etc.

To change the default device back, type `V` again and enter `D:` at the prompt.

Even though the utility functions do not require much memory, advanced users might want to include them in a customized menu definition, see section 12.12. The name of the utility item group containing all the above items is `UTILITY.MEN`.

12.10 Configuring Dos++

The last two menu items `W` and `X` allow you to view the current Dos++ configuration, and to customize Dos++. Customization data will be written to a file named `CONFIG.SYS`, which will restore the settings once the system boots up from the disk containing the file. The customization options the menu offers allow you to specify the number of file and disk buffers FMS++ allocates from memory, and hence how much memory will be available to other programs and cartridges such as BASIC.

If the FMS++ overlay manager is present, see section 13, then customization provides little benefits as all FMS buffers will be placed under the BASIC ROM with no byte of BASIC program memory spend for FMS++ anyhow. Since then FMS++ buffers do not use BASIC memory, the overlay manager already installs the largest possible configuration, without compromising available memory. Customizing the number of buffers thus provides only a benefit if you are working with machine language programs or cartridges that do not include a bank switching logic, i.e. cartridges that make the overlay manager non-functional. Thus, in general, customization is rarely needed, and only required for expert usage.

To start customization, press `W` on the keyboard. The first dialog allows you to modify the number of file buffers FMS++ allocates from RAM:

```
File buffers        (1-8):3
```

Each open channel requires one file buffer. Thus, with the default number of three file buffers, FMS++ can open at most three files at once. Whether the file is open for writing, reading, updating, or reading the disk directory does not matter. As a rule of thumb, three buffers is sufficient for most programs. Each buffer requires 128 bytes of memory, unless the overlay manager is active. In the latter case, the number of file buffers is already set to the maximum of eight.

The second configuration to adjust is the number of drive buffers:

```
Disk drives          (1-8):2
```

Each disk drive FMS++ administrates requires exactly one drive buffer. That is, with the default of two disk drive buffers, only disk drives one and two can be used, and drive three will be unavailable. If you have more drives you want to make available, increase the number here. Again, each drive buffer requires 128 bytes of memory, unless the overlay manager is in use. In that case, no extra memory is required and the number of drive buffers is set to four. Larger numbers are possible, but rarely used in practice since the Atari 1050 drives can be only configured to units one to four, and more than four drives are not supported with at least this drive type. The default value is two here.

The next question is to whether enable write verify or not. If enabled, the disk drive includes a verification step for each sector written, and thus ensures that each and every byte was really successfully written to disk, at the price of making disk writes half as fast as disk reads. Unless the drive or the disks you use are very weary, write verification provides little benefits as the Atari drives are in general quite reliable, and thus this option is off by default. You may enable it here by pressing `Y`. Alternatively, you may also enable it on a file by file basis by using the write-verify modifier `/V` on the file name, see section 8.11:

```
Write with verify (Y/N):N
```

Finally, Dos asks you to insert a boot disk into the first drive. The configuration data will be written to this disk, and will become active as soon as you boot from this disk. *Any changes to the configuration will be pending and inactive until you reboot the system with this disk inserted in the first drive*. You may now abort the operation by pressing `BREAK`. Dos assumes that the disk you insert now is already formatted, it *will not attempt to format the boot disk*. If you need to format the disk, abort at this point and use the menu item `I` for a low-level disk format to prepare the disk.

If Dos++ finds a configuration file already present on the disk, it will ask whether it shall replace or append the new configuration data to this file:

```
CONFIG.SYS already exists.
Append to or Ovewrite with new
configuration (A/O):A
```

The `O` option completely removes the old configuration file, and in specific, will also remove the overlay manager (see 13) from the disk, if any is present, or any other custom configuration you may have made. The `A` option will append the new configuration at the end of the old. If the currently present `CONFIG.SYS` file was also created by menu item `W`, this provides no benefits, the file will only grow longer and the new configuration will replace the old one. With custom written configuration files, this may or may not merge the new configuration with the functions of the already present configuration file. *Settings of the overlay manager cannot be adjusted by appending to it.* As a rule of thumb, the Dos configuration should be left alone if the overlay manager is active, or any hand made, non-Dos written `CONFIG.SYS` is present. Otherwise, option `O` should be preferred, unless you know what you are doing. In case you loose the overlay manager by replacing it by your configuration: The system disk contains a copy of it as `FMSOVL.EXE` - copy this file over `CONFIG.SYS` to restore it.

The `X` menu item lists the currently active configuration, and some additional information that is mostly relevant for experts. This menu item does not take any input, but rather prints the configuration and returns immediately. The following lists an example output with the overlay manager active:

```
File buffers       :8
Number of drives   :4
Write with verify  :off
Free memory        :18770
Application memory :37824
Disk buffers at    :$A100
```

The information printed here reads as follows: The first and second number are the number of drive and file buffers, respectively. They can be adjusted by the W menu item, unless the overlay manager is active. The number of file buffers defines how many files FMS++ can open simultaneously, the number of drive buffers how many disk drives FMS++ can manage; each buffer takes 128 bytes of RAM, unless the overlay manager moves them under the BASIC ROM. Write with verify can be either on or off, and if it is on, the disk drives apply an additional verification step after having written each sector — which makes writing more secure, but much slower.

"Free memory" displays the number of bytes available for Dos++ for copying disks or files. All other memory is taken by Dos and the menu. The larger the number here, the less often you need to swap disks when copying files or disks. The number below shows the number of bytes BASIC or any other program may allocate from the operating system when the menu becomes inactive again. It is the available system RAM, minus the RAM required by the operating system, minus the RAM required by FMS++. The RAM allocated by the menu is not counted here. The BASIC function `FRE(0)` may show a somewhat smaller number even if no program is loaded since BASIC itself also requires a bit of RAM for itself.

The last number shows the address in hexadecimal where the first disk buffer is located in RAM. This is also the start of the FMS++ buffer area. From this address upwards FMS++ requires $(d + f) \times 128$ bytes, where f is the number of file buffers and d is the number of drive buffers. All addresses beyond file and drive buffers are available for BASIC or application programs. If the overlay manager is present, the disk and file buffers are relocated behind the BASIC memory, and hence a rather high address is show here, as in the example. Then, only a small number of bytes is required for the overlay manager itself.

If you want to include the configuration item in a customized menu, the this item group contained in the file `CONFIG.MEN`.

12.11 Additional Features: Limited DOS 3 Support

The Dos menu also includes a functionality to read files from Atari DOS 3, which never became very popular. Very much like DOS 3, the Dos menu allows you to convert files back from DOS 3 to FMS++, though in a less kludgy and more elegant way than the forwards conversion offered by the Atari DOS. Once you load the DOS menu, a new device is linked into the system that allows read-only access to DOS 3 formatted disks. The name of this device is `3`, the digit "3".

To list the contents of a DOS 3 disk, insert the disk in the first drive, select the menu item `A`, press `RETURN` *once* and enter the following as file specification:

```
3:
```

The result will be the disk directory of the DOS 3 disk.

To convert files from DOS 3 to FMS++, have the original DOS 3 disk ready, and provide a FMS++ formatted disk to which you want to convert the files to. If you have only a single disk drive, insert the DOS 3 disk in the drive, select menu item `O` and enter the following as source and destination for the file copier:

```
Duplicate - give from(,to) :
3:*.*/I,D:
```

This will copy all files from the DOS 3 formatted disk to an FMS++ disk. The `/I` modifier informs the disk copier that the source and destination are actually on the same physical hardware and you need to swap disks to copy between them. Then, follow the instructions on the screen to swap disks in and out as the conversion proceeds.

If you have two disk drives available, insert the DOS 3 source in the first drive, the target disk into the second, and provide the following input to item `O`:

```
Duplicate - give from(,to) :
3:*.*,D2:
```

If you only want to copy a single file, or a subset of all files, provide the appropriate wild-card behind `3:`, see section 8.9.

The DOS 3 device will become unavailable as soon as you leave the Dos menu, and as soon as you run an external command. This is because the DOS 3 device is actually realized as an menu item group of the DOS menu which, quite surprisingly, does not include a single item but just installs the DOS 3 emulation.

The DOS 3 emulator is also able to support multiple drives, but can only open a single file at a time. A DOS 3 filing system on the second disk is addressed as `32:`, and — you guessed right — on the third drive as `33:`. The first digit "three" is the name of the device, the second enumerates the drive units. Last but not least, if you want to include the DOS 3 handler in a custom menu configuration, see the next section, its item group file is named `FMS3.MEN`.

12.12 Advanced Information: Customizing the Menu

Unlike all other DUPs, the FMS++ menu can be fully customized. That is, groups of items can be placed in any order, and included or excluded as you wish. In the configuration that comes on the system disk, the menu includes the full set of available items and hence takes longer to load and requires more memory than necessary. By customizing the menu, you can shrink it down to the set you need to work with, and can still run the excluded menu items when needed as "external items" as described in section 12.1. Configuration is done by the file `MENULST.SYS`, which can be either modified or created by an editor or a short BASIC program. This file lists, one line at a time, the names of the item group files to include in the menu when starting it.

The following text provides a step by step instruction how to configure the menu to include the "minimal useful" set of commands. Of course, your preference may vary and you may want to include other items in the list, feel free to modify as you wish.

First, it is a good idea to make a backup of the current `MENULST.SYS` file in case something goes wrong. If you are right now in the menu, use the menu item `O` to duplicate a file, and enter

```
MENULST.SYS,MENULST.BAK
```

Otherwise, from the DOS command line or DiskIO in BASIC, type

```
COPY MENULST.SYS,MENULST.BAK
```

to copy the file to a backup.

The next steps require you to write a small BASIC program that writes a new `CONFIG.SYS` file. Thus, exit the menu or the Dos command line to BASIC now, clear the BASIC memory by entering `NEW`, and type in the following program:

```
10 OPEN #1,8,0,"D:MENULST.SYS"
20 ?#1;"DIR.MEN"
30 ?#1;"XIO.MEN"
40 ?#1;"FORMAT.MEN"
50 ?#1;"BINARY.MEN"
60 ?#1;"DUPDISK.MEN"
70 ?#1;"DUPLICAT.MEN"
80 CLOSE #1
```

This will include the menu items `A` through `O`, and will replace item `P` with the disk duplicator. What this program does is that it re-creates a `MENULST.SYS` file where each line in the program, and hence each line in the file lists one menu item group to be loaded in the full menu. The `DIR.MEN` item group includes the commands to list the disk directory, leave the menu to BASIC or to the command line. `XIO.MEN` includes delete, rename, lock and unlock functionality, `BINARY.MEN` the binary load and binary save functions. If

you don't need them, leave out line 50. Line 60 includes the file duplicator, and line 70 the disk duplicator. This is approximately the features you get with the standard Atari DOS. Utility functions, DOS 3 support, Dos configuration and Os/A+ emulation are then left out and do not require memory. Yet, you can still load them if you need them by the key which loads external menu items, see 12.1.

13 The Overlay Manager

The overlay manager is a FMS++ system component that releases even more RAM to BASIC and other cartridges by moving the disk buffers under the otherwise unavailable RAM portions under the cartridges. With the overlay manager active, FMS++ requires only a very minimal amount of 256 bytes of BASIC Ram instead of the 4K a regular DOS would require. The overlay manager comes in the form of the `CONFIG.SYS` file on the system disk that is loaded whenever the system disk is inserted in the first disk drive, and the drive is turned on before the main computer system. The FMS++ bootstrap mechanism will then attempt to locate this file, and by that load the overlay manager into the system.

Since the overlay manager uses RAM under the cartridge area to relocate the disk and file buffers to, it requires a cart to be inserted, or the BASIC to be active. Additionally, the cart must provide a mechanism to be "banked out", i.e. disabled, such that FMS++ can gain access to its buffers. The overlay manager currently supports the banking mechanism of the XL BASIC ROM, and that of Oss switching cartridges such as Mac/65. If no cartridge is found, or the overlay manager is unable to determine the banking mechanism of the cart, it automatically disables itself and disk buffers will be placed in the low system RAM area as usual.

To enable better RAM usage on other disks, copy the `CONFIG.SYS` file from the system disk to any other disk, and boot from this disk. In case the file gets damaged or corrupted, the system disk contains an identical copy of it under the name `FMSOVL.EXE` that can be copied over to `CONFIG.SYS` by the Dos command line or the Dos menu.

13.1 Advanced Information: The Overlay Manager API

The overlay manager provides a simple API to allow programs to enable and disable banking. The API consists of the following functions in page 7:

```
$075E     SWITCHOFF
$076A     SWITCHON
```

The first function banks the cartridge off and enables access to the RAM under the cartridge. It also takes care of storing the currently active bank of the cartridge. The second function maps the cartridge in again, and restores it to the bank that was found active when banking it out.

Availability of these two functions shall be tested by the following procedure:

```
$074C   FMSBANK
```

If this RAM location contains the byte value `$20`, and the following two locations contain the low- and high byte of `SWITCHOFF`, i.e. the values `$5E` and `$07`, then availability of the overlay manager is granted. Any other values indicate that the overlay manager is not available, and no attempt should be made to call either `SWITCHON` or `SWITCHOFF`.

14 Dos++ Binary Load File Format

The Dos++ file format for binary (machine language) programs is identical to the binary format specified by Atari DOS 1, DOS 2.0S and 2.0D, DOS 2.5 and DOS 3. For reference, its specification is reproduced here.

The first two types of a binary load file must be `$FF`, indicating a binary file, defining its *header*. The remaining structure of the file defines regions or blocks of memory, to be loaded from disk into RAM.

Each block starts with the start address, i.e. the first byte to load, first the low, then the high byte, and its end address, again first the low and then the high byte. Both start and end address are inclusive, i.e. the end address is the *last byte that is still included in the following block of data*. The raw block data of `END-START+1` bytes follows immediately the block header. Once a block is completed, the file may either end, or may continue with a new block, start and end address, plus block data. The second and later block may, but need not include the two `$FF` bytes. The following binary file consists of a single block that defines the contents of page 6:

```
FF
FF      these are the header bytes
00
06      start address, page 6
FF
06      end address, last byte to be included at end of page 6
...     256 bytes follow
```

The possibility to include a next header, followed by block data, allows to built-up a binary load file containing multiple memory regions sequentially by appending at the end of an already existing file with the `/A` modifier of FMS++. The following example commands for the Dos command prompt would first save the contents of page 6, and then append the memory region from `$1F00` to `$3000` in the file:

```
SAVE PROGRAM.COM,0600,06FF
SAVE PROGRAM.COM/A,1F00,2FFF
```

Two special memory regions exist that, when defined by a binary load program, trigger additional action by Dos++ when loading the file:

The two bytes at `$2E0` and `$2E1` carry the run address of the file Dos will jump to as soon as the program has been loaded *completely*. Running a program through this vector can be inhibited by specifying the `N` modifier.

The two bytes at `$2E2` and `$2E3` define an initialization vector that is jumped to *immediately after* it has been defined. That is, a program may contain several initialization vectors, and loading a file from Dos will call all of them as soon as they are encountered, but only one run vector that is only honored at the very end of the loading process. When the run vector is defined multiple times, only the value defined last is used.

The following example defines a program in page 6 that is initialized at address `$630` and run at address `$600`:

```
FF
FF      these are the header bytes
00
06      start address, page 6
FF
06      end address, last byte to be included at end of page 6
...     256 bytes follow
E2
02
E3
02      This defines the init vector
30
06      Init vector value: $630
E0
02
E1
```

```
02      This defines a block containing the run vector
00
06      Run vector value: $600
```

Unlike in the example, it is *not necessary* to define run and init vector in separate blocks. They may also be merged into a single block defining the memory region from `$2E0` through `$2E3`.

Last but not least it should be mentioned that this file layout is *not completely* compatible to that defined by Os/A+. While the syntax is identical, Os/A+ defined the convention that even files *without* a run vector are run, namely from the first start address of the first block contained in this file. Dos++ *does not* implement this convention, except for the `Load Os/A+` menu item in the Dos menu. Thus, Dos++ follows the Atari DOS conventions and not the Os/A+ conventions.

It is furthermore suggested for disk based device handlers to use the *init vector* and not the run vector to install the handler into the operating system. This allows users to concatenate several handler files into one big file, and thus simple installation of the handlers through one single binary file.

14.1 Dos++ Item Group Files

While the Dos++ menu is itself a binary load file, the *item group files* ending on `.MEN` *are not*. They have their own structure which define a *relocatable* binary file. Such files allow free placement of program data in memory, and hence allow the dynamic composition of the Dos++ menu.

Similar to binary files, menu item groups also have a two-byte header, though the header is in this case twice the byte `$9B`.

What follows are the definitions of the menu items in the menu group. Each menu item is defined by a two byte offset from the start of the body that defines the entry function to call when the menu item is invoked, and an exactly 18 bytes of text, encoded in the internal ANTIC code defining the name of the menu item that appears on the screen. The offset is encoded with the low-byte first, followed by the high-byte. The list is terminated by two zero bytes, i.e. if the offset is zero. An 18 byte text is not included for the last dummy entry.

The following two bytes immediately after the menu item list define the offset from the start of the body to the init function that is called once the menu item has been loaded. The init function must be present, but may simply point to a single `RTS` instruction in case it is not needed.

The next two bytes define the length of the actual body that contains the machine code for all menu items in the file, with the actual body following; this body contains the actual machine code. However, the machine code should be built such that the first byte of the body has the absolute address zero. The menu takes care of relocating the body to its final position, which will always be at the lowest available page (256 bytes) boundary.

Relocation information follows; this information allows the menu to relocate the item group to its final location in RAM. The relocation information consists of $\lceil l/8 \rceil$ bytes, where l is the length of the body in bytes, as indicated by the two bytes preceding the body. This information is a bit mask, with bits sequenced from MSB to LSB in each byte. The bit at **bit-offset** o from the start of the bit-mask defines whether the byte at **byte-offset** o from the start of the body requires relocation. Since the body is always relocated to the start of a page and is assembled to offset zero, only high-bytes in the body require relocation and a single bit-mask is sufficient to specify which bytes of the body to modify. In specific, if the bit at bit-offset o is set, then $[d/256]$ is added to byte o of the body, where d is the relocation destination.

The following table summarizes this format:

```
$9B
$9B             menu item group header
O1 mod 256
O1 div 256      entry offset of the first item
```

```
TTTTT....       18 bytes item name, ANTIC coded
O2 mod 256
O2 div 256      entry offset of the second item
TTTTT....       18 bytes item name, ANTIC coded
etc...
00
00              end of item list, two zeros
I  mod 256
I  div 256      init offset, lo and hi byte
L  mod 256
L  div 256      length of body, lo and hi
BBBBB.....      L body bytes, machine code
                offsets count from the first byte
                of the body
RRRRR....       ceil(L/8) bytes of relocation
                information
```

14.2 The Menu API

While the command line interface API for external programs is minimal, the Dos++ menu offers a rich API and a huge set of support functions that are documented here. As soon as the menu is loaded, the menu API interface is installed into page 6 and can be called from there from any menu item group file. Note well that programs cannot make permanent changes to this API, i.e. any changes will be overwritten as soon as a menu item returns back to the menu.

`$600 : DUPEND` These two bytes store a pointer to the first free memory byte available for buffering. It thus points behind the end of all loaded menu item groups. Programs should, however, allocate memory from the heap via `RESERVE` instead.

`$608 : DEFAULTDEVICELEN` Contains the length, in characters, of the default device. Its value is typically three: Device character, unit number and colon.

`$609 : DEFAULTDEVICE` The menu default device to operate on if the user specifies no device. Unless changed by the user, this will be `D1:`. Note that you *cannot* change the default device by overwriting these three bytes. Instead, use the API `SETDEFAULDEVICE`.

`$612 : PRINT` Print the string at the address given by the register pair X,Y, low in X, high in Y, number of characters is in the accumulator.

`$615 : SETIOCB` Set the channel for miscellaneous IO operations. The IOCB offset, i.e. the channel number pre-multiplied by 16, shall be specified in the X register.

`$618 : LOADINPUTBUFFERPTR` Load the register pair X,Y with the pointer to the user input buffer, and the A register with its length. This menu provided buffer is used for any line-oriented input from the user. On return, the X register contains the low-byte and the Y register the high-byte of the buffer address. The A register contains its length. Note that the position of this buffer may change as the menu is fully relocatable.

`$61B : LOADBUF1PTR` Load the register pair X,Y with the address of the first output buffer, and load A with its length. This is one out of three output buffers pre-allocated by the menu. Its suggested use is to store user input temporarily for parsing or output. Note that the position of this buffer in memory may change, and this function shall be called to obtain its position.

`$61E : LOADBUF2PTR` Load the register pair X,Y with the address of the second output buffer, and A with the length of this buffer.

$621 : LOADBUF3PTR Load the register pair X,Y with the address of the third output buffer, and A with its length.

$624 : OPEN Open an IOCB channel with the channel number specified by SETIOCB, to the file specification pointed to by the register pair X,Y and the open mode in A. X contains the low and Y the high-byte. The AUX2 value of the open mode will be set to zero, AUX1 comes from A. Returns an error code in the Y register.

$627 CLOSE Close the IOCB channel set by the last call to SETIOCB. Returns an error code in the Y register.

$62A XIO Run a generic CIO command on the file specified by the X,Y register pair, the command itself comes from the A register. The IOCB channel comes from the last SETIOCB call. AUX1 and AUX2 will remain unchanged and need to be set by the user before calling this, X contains the low and Y the high-byte of the file specification. Returns an error code in the Y register.

$62D CLEARAUX Resets AUX1 and AUX2 in the IOCB channel set by SETIOCB. Should potentially be called before XIO to set the AUX values to defined values.

$630 GETLINE Reads an EOL terminated line from the terminal into the buffer pointed to by X,Y. The buffer length is given by A. In case of an error, this call returns directly to the menu main loop and does not return to its caller. The suggested usage is to first call LOADINPUTBUFFERPTR to load registers with the menu input buffer, and then this call to collect input from the user.

$633 ERROR Signal the IO-error in Y to the user, release all allocated memory, close all open channels and return the menu main loop. Does not return to the caller. This call will also print a clear-text description of the error for many typical error codes.

$639 CLEARHEADLINE Clears out the line between the menu item list and the user input area that typically displays notes or error codes.

$63C RESERVE Allocates bytes from the system heap. The number of requested bytes is passed in in the register pair X and Y with X containing the low-byte and Y the high-byte. The address of the reserved memory block is returned in the same register pair with the same conventions. If the system runs out of memory, an error is signaled and the program flow returns to the menu main loop. Memory is released automatically as soon as the program returns to the menu loop.

$63F DISPOSE Releases all memory from the address X and Y onward to the system heap. The X register contains the low and the Y register the high-byte of the memory to be released.

$642 RELEASEALLMEMORY Releases the entire system heap back to the system.

$645 CHECKIOERROR Checks the error code in the Y register. Returns with the carry flag set on an EOF, carry flag cleared on no error, or directly calls the error handler of the menu, aborting the current program flow.

$648 ABORTONBREAK Checks the error code in the Y register. Returns with the carry flag cleared on no error, with the carry flag set on error, and returns to the menu main program loop if the BREAK key was pressed.

$64B BLOCKDISPLAY Disable the menu, switch to the textual output and block the user from displaying the menu.

$64E READINPUTPARAMETER Read the next command argument up to the next comma from the input buffer to the buffer pointed to by the register pair X and Y. Returns with the zero flag set if the parameter is empty, and with the carry flag set if the end of the line has been found and no further argument could be parsed off from the input buffer.

$651 COPY Copy the bytes pointed to by the register pair X and Y to the address contained in FR0 and FR0+1. The number of bytes to be copied is passed in the accumulator. FR0 is here the first floating point register on the zero page documented in the memory map in section 21.

$654 PARAMETERERROR Signal an error with one of the parameters to the user and return to the main program loop.

$657 SETREGULARCOLOR Installs the regular color theme on the screen, namely blue. This color theme should be used if nothing special happens and the menu expects user interaction.

$65A SETREDCOLOR Install the warning color theme on the screen, namely red. This color theme should be used if the user could trigger a potentially dangerous activity.

$65D SETGREENCOLOR Install the green color theme that should be used if user input could trigger a harmless reading operation, for example reading the source(s) of a copy operation.

$660 SETBLUECOLOR Install the reserved color theme. Currently not in use, and could be used for operations out of the ordinary.

$663 SETYELLOWCOLOR Install the error color theme. This theme is used by the system to signal an error.

$666 UNBLOCKDISPLAY Reverses the effect of BLOCKDISPLAY and returns the control on the menu visibility to the user. The state of the menu display does not change.

$669 DISABLEBREAK Locks the BREAK key, i.e. disables it.

$66C ENABLEBREAK Re-enables the BREAK key.

$66F TODECIMAL Converts the unsigned number in the register pair X and Y, with the low-byte in X, into a decimal ASCII string in the third output buffer. The buffer is filled completely, spaces are used to left-align the number into the buffer.

$672 TOHEX Converts the unsigned number in the register pair X and Y, with the low-byte in X, into its hexadecimal ASCII representation in the third output buffer. The buffer is filled completely, spaces are used to left-align the number into the buffer.

$675 FROMDECIMAL Converts the ASCII encoded decimal number in the buffer pointed to by the register pair X and Y into a binary number in registers X and Y. The low-byte is in register X. Triggers a parameter error if the number is ill-formatted.

$678 FROMHEX Converts the hexadecimal number in the buffer pointed to by the register pair X and Y into a binary number in the same registers. The low-byte is in register X. Triggers a parameter error if the number is ill-formatted.

$67B SETAUX Installs the value in register X into AUX1 and the value in register Y into AUX2 of the currently selected IOCB channel; selecting an IOCB works via SETIOCB.

$67E SETIOCBADR Installs the address in the register pair X and Y with the low-byte in Y as the buffer pointer of the currently selected IOCB channel.

$681 SETLENGTH Installs the length in the register pair X and Y with the low-byte in Y as the length of the currently selected IOCB channel.

$684 SETCOMMAND Install the CIO command in the accumulator into the currently selected IOCB.

$687 RUNCIOCMD Run the CIO command in the accumulator with all other parameters already installed through the currently selected IOCB channel. Does not test for errors, this is up to the caller. The error code is returned in the Y register.

$68A GETKEY Read a single character from the keyboard without echoing it on the screen and return it in the accumulator.

$68D BGET Read a block of characters from the currently selected IOCB channel to the buffer pointed to by the register pair X and Y. The low-byte is in the X register, the length of the buffer must have been already installed into the channel. The error code of the operation is returned in the Y register.

$690 BPUT Write a block of characters to the currently selected IOCB channel, the pointer to the buffer is passed in the register pairs X and Y with the low-byte in X. The length of the memory block must have been installed into the IOCB already. The error code is returned in the Y register.

$693 PRINTRECORD Print the EOL terminated string pointed to by the register pair X and Y. The maximal length to be printed is passed in the accumulator. The low-byte of the pointer to the string is in register X.

$696 ADDDEVICE Complete the file name pointed to by the register pair X and Y by adding the default device name to it if the file name does not yet include a target device specification. The low-address of the file name buffer is in register X.

$699 CONVERTDIRECTORYENTRY Converts the output of a Dos 2 compatible directory listing into a canonical file name by separating name from extender and inserting a dot between them. The directory entry starts with the *file locked* indicator, the third byte in the buffer is the start of the file name. The input buffer is pointed to by the register pair X and Y with the low-byte in X. The output is placed into output buffer 3.

$69C GETARGUMENTEXTENSION Test whether the file name or command argument in the buffer pointed to by the register pair X and Y contains the extension given in the accumulator. The extension is separated from the file name with a forwards slash . If found, the carry flag is set and the extension is removed. Otherwise, the carry flag is cleared. The low-byte of the input buffer address is in register X.

$69F COPYBUFFER2BUFFER Copy the buffer with index in the X register to the buffer of the index given by the Y register. Buffer zero is the input buffer, buffers one to three are the first to third output buffers.

$6A2 CURSOROFF Disable the text cursor.

$6A5 CURSORON Enable the text cursor.

$6A8 PRINTEOL Print a line feed to the screen.

$6AB SETZPTR Load the menu ZPTR at address $B0,$B1 with the values in X and Y. The low-byte is in register X, the high-byte in register Y.

$6AE RUNCARTRIDGE Exit the menu to the inserted cartridge or to the DOS command line if no cartridge is found.

$6B1 RUNCOMMANDLINE Exit the menu to the DOS command line.

$6B4 YESNO Wait for a single key press from the user. Returns with the carry set if the user answered Y, returns with the carry cleared in all other cases.

$6B7 CLOSEALL Close all IOCB channels except channel zero connected to the editor.

$6BA GETDIRECTORYLIST Returns a list of all file specifications matching the pattern pointed to by the register pair X and Y with the low-byte in register X. Wild-cards are resolved and the default device is pre-pended to the wild-card if it does not include a device name. The list of matching file names is allocated on the heap, and the pointer to the first match is returned in the register pair X and Y with the low-byte in X. Each entry is 32 bytes long, the first byte is a flag byte that is zero initially or has bit 7 set if the entry has already been worked on. The file name follows, which is EOL terminated. Additional status bytes can be included by GETBUFFEREDLENGTH and SETBUFFEREDLENGTH (see below). The end of the list is indicated by a file name that consists of a single EOL.

$6BD GETDIRECTORYENTRY Load the next available entry from the directory list pointed to by the register pair X and Y and copy the file specification into the input buffer. The low-byte is in the X register, the high-byte in the Y register. Returns with the carry flag set if the end of the list has been reached and all entries have been worked on, returns with the carry cleared otherwise. For backwards compatibility reasons, the register A should be set to the value contained in DEFAULTDEVICE+1, the device unit. This value is ignored by the current version as the directory list contains completely specified file names including device names, but older versions added the default device unit (not name) when retrieving list entries. Available entries are indicated by a positive flag byte; the first byte of the 32-byte entry must have bit 7 cleared to qualify an entry. Setting this bit and hence removing entries from the list is up to the caller.

$6C0 GETUNITNUMBER Extracts the unit number from the device specification in the file specification pointed to by the register pair X and Y, with the low byte in X, and returns the ASCII number in register A. This call should not be used anymore as menu items should prefer to work with full device names and not only unit numbers of the disk handler.

$6C3 SETDEFAULTUNIT Sets the default device to the disk device whose unit number is given in register A. This call should not be used anymore, instead programs should specify full device names as default devices by means of SETDEFAULTDEVICE.

$6C6 GETLENGTH Extract the IOCB length returned by the last CIO command, from the IOCB selected by SETIOCB. Returns the length in register pair X and Y with the low-byte in X.

$6C9 RESTOREOSDISPLAY Remove the DOS menu from the screen and restore a regular text screen.

$6CC INSTALLDISPLAY Re-install the DOS menu on the screen.

$6CF INSTALLDESTRUCTOR Installs a call-back to address X, Y with the low-byte in the X register that is called as soon as the current menu item is removed. This destructor should remove any installations or hooks the menu item performed. The destructor itself shall consist of two jump-instructions, the first going to the actual destructor method of the menu item. This method shall *not* return by RTS, but instead then jump into the second (subsequent) jump instruction of the destructor. This second jump instruction shall be assembled to go to $E4C0 which contains an RTS instruction. The purpose of these two jump instructions is that the menu can by this mechanism chain several destructors together which then call automatically one after another. The second JMP instruction is patched over by any subsequent INSTALLDESTRUCTOR calls to return control to the destructor registered earlier.

$6D2 REMOVEMENU Removes the menu display through RESTOREOSDISPLAY and re-installs the regular DUP vector. This call should be made before running external binaries to ensure that they receive a proper text window and an OS interface without the menu.

$6D5 INSTALLMENU Re-installs the menu display through INSTALLDISPLAY and sets the DUP vector to point to the menu such that a reset returns control to the menu and not to the command line. This should be called when returning from an external binary to re-establish the DOS menu.

$6D8 GETDEFAULTDEVICE Copy the default device specification to the buffer pointed to by the register pair X and Y, with the low-address in register X.

$6DB SETDEFAULTDEVICE Install the default device from the buffer pointed to by the register pair X and Y, with the low-byte in register X. If the buffer did not contain a valid device, this call returns with the carry flag set. Otherwise, returns with the carry flag cleared.

$6DE NEXTDIRECTORYENTRY Advance the zero pointer register pair whose address is given in the X register by one entry in the directory list. Does not skip over already busy or used entries. Returns with the carry flag set if the last entry in the directory list has been reached.

$6E1 GETBUFFEREDLENGTH Return a (user-defined and user interpreted) buffer length (16-bit integer) from the directory buffer pointed to by the zero page register $D6, $D7. Return the integer in register pair X and Y, with the low-byte in X.

$6E4 SETBUFFEREDLENGTH Keep a 16-bit integer, typically a buffer length, as side information in the directory entry pointed to by the zero page registers $D6, $D7. The 16-bit integer is passed by the register pair X and Y, with the low-byte in register X.

$6E7 to $6FF Reserved for future extensions.

15 The Tape Handler

While Os++ contains a tape handler under the device name C, this device is only a placeholder and does not function. Any attempt to read or write data to the tape will generate an error. The tape handler had been off-loaded to disk, and can be loaded from there instead when required. Since the tape handler requires some RAM by itself, and thus requires BASIC memory, make sure you save your programs first before continuing as loading the tape handler will erase them.

First, exit from BASIC or the cartridge to the Dos command line by typing in

```
DOS
```

and pressing RETURN. Then insert the system disk, and load the tape handler with

```
D1:TAPE.EXE
```

then, finally, return to BASIC with

```
CAR
```

The tape handler is now loaded and active. BASIC commands CSAVE and CLOAD will be operational again, though a bit of BASIC memory is lost.

15.1 Tape Handler Extensions

While the tape handler is disk based, it has been somewhat extended in functionality while staying completely backwards compatible. Similar to FMS++, it also supports *modifiers* in its file specifications that influences its behavior.

While BASIC programs saved to tape with CSAVE use a short inter-record gap that loads and saves somewhat faster, the alternative LOAD "C:" and SAVE "C:" commands accept and create an incompatible format with a longer inter-record gap and hence take longer. The /S modifier switches both commands to the *short block mode* and hence create a compatible format:

```
SAVE "C:/S"
```

is thus identical to CSAVE, and LOAD "C:/S" to CLOAD.

In principle, the /S modifier could also be applied to LIST and ENTER, but since loading programs by ENTER is slow and the tape does not stop for the short-inter record gap, chances are that loading a file created by LIST with the short-inter record gap fails. The short format is thus not recommended for LIST or ENTER.

In addition to the short block mode triggered by /S, the /T mode starts a "turbo" mode that loads and saves slightly faster. Tape records in this mode are twice as large, creating only half of the gaps, and the tape transmission baud rate is slightly increased. Since CLOAD and CSAVE do not take file names, the turbo mode works only in conjunction with LOAD and SAVE:

```
SAVE "C:/T"
```

saves the currently loaded program to tape, using a slightly increased speed. Note, however, that the turbo mode is incompatible with the regular tape mode, and you need to remember which file you saved in turbo mode and which you did not since the parameter when loading needs to match the one used when saving the file.

The following table lists all modifiers supported by the tape handler:

Table 7: Tape Modifiers

Modifier	Function
S	Enable short inter-record gap
T	Enable turbo mode
N	Load, do not run (see next section)

15.2 Booting from Tape

Since the tape handler is no longer ROM resident, booting from tape is not available by traditional means. In specific, holding the START key while turning on the machine yields nothing. However, the resident tape handler implements an emulation that works as long as the program to bootstrap does not overlay with the tape handler in RAM.

For that, first enter DOS, insert the tape to load from and rewind to its initial position. Then type the following at the Dos command prompt:

```
C:-/S
```

this will attempt to load an "external Dos command" from tape by means of the "binary load" handler command. Instead of accepting the binary load format of Dos files, it reads however tape boot files and runs them. The modifier /N suppresses the program start-up, but not program initialization, similar to the FMS /N modifier specified in section 8.11. Thus, the command line

```
C:-/S/N
```

at the Dos command prompt will attempt to load a tape boot file, will run its initialization function, but not its run function.

15.3 SIO Tape Support - Information for Advanced Users

Unlike the Atari ROMs, the serial interface routine SIO of the Os is no longer capable of initiating communications to the tape. This is because the tape is, unlike all other SIO devices, a "dumb" device and does not follow the SIO protocol spoken by the intelligent devices on the SIO chain. The disk-based tape handler implements the corresponding replacement functions separately.

16 Central I/O

The CIO call of the operating system is the central call-in for all channel-based communications to the device handlers installed in the system. In BASIC, CIO is used by the `OPEN` and `CLOSE` commands, by `GET` and `PUT`, for drawing lines or plotting points. BASIC users usually don't need to worry about CIO as BASIC establishes all necessary communications to the device handlers through this part of the operating system.

16.1 Information for Advanced Users

The Os++ CIO differs in several minor points from the CIO routine in the Atari ROMs. Despite not offering support for parallel-port devices never manufactured by Atari itself, it also differs slightly in details of the CIO `OPEN` command:

The Os++ CIO first reads the `Put` vector of the device handler, installs this vector into the `PutOneByte` vector of the IOCB, and then calls the handler. The `Open` vector of the handler has now the possibility to replace the `PutOneByte` vector by its own specialized vector, and thus to capture software that — actually incorrectly[2] — calls this vector to output data, rather than to go through CIO as it should. That is, handlers can actually supply *two* instead of one `Put` vectors: One for BASIC, and one for CIO. The resident disk handler uses this feature to enable high-speed I/O ("bursting") only when called from CIO, instead of second-guessing from the return address whether bursting is possible or not.

Unfortunately, one insanity remains for backwards compatibility reasons: If a CIO `OPEN` call fails, the corresponding channel is not automatically closed. It remains still connected to the handler and an explicit `CLOSE` command should be issued to indicate the handler that it should release all resources associated to this particular channel.

17 Serial I/O

The SIO or serial I/O call of the operating system sits several layers below CIO and initiates transports over the serial bus of the Atari computer. For example, the printer handler does not talk to the printer directly, but uses SIO to transport the data from the printer buffer over the serial port to the printer. SIO cannot be called from BASIC directly, and ultimately, only handlers and low level system services should call SIO to implement their functionality. SIO is for advanced users only.

17.1 Information for Advanced Users

Unlike the Atari ROMs, the SIO in Os++ no longer includes the functionality to generate two-tone modulated data as required by the program recorder, and is neither able to read data from the tape. Communications to the tape is rather unorthogonal, and its ROM space was required for more important features. The disk-based tape handler includes a SIO emulation instead.

Serial communication differs only slightly from the original Atari SIO: While the external protocol stays identical, the Os++ SIO initiates serial transport in a slightly different way: While the Atari ROMs initiated transport by writing into the serial port register of POKEY directly, the Os++ SIO is entirely interrupt-triggered and SIO depends now on the POKEY *Data Register Empty* interrupt to start the transfer. This has the advantage that user programs can hook into the SIO interrupt handler, and replace it, for example to transfer data from the RAM under the Os-ROM.

[2] like BASIC

18 The Disk Interface Vector

Even though the Atari ROMs did not include a file management system, they did include a "convenience vector" for simplified disk communication that sits just one level above SIO. The so-called `DiskInterf` vector only interprets the SIO commands and fills the target device, communication direction and, for some commands, the buffer pointer accordingly to talk to disk drives. `DiskInterf` makes disk communication slightly more convenient, but without including the ability to administrate files on disk. It can only address individual disk sectors, read them, write to them, get the disk status, or format the disk. The `DiskInterf` vector is not directly available from BASIC.

18.1 Information for Advanced Users

`DiskInterf` has been extended to support all *official* commands of the 1050 drive, that is: Read sector, write sector with and without verify, drive status, and format in single or enhanced density. In fact, the ROM based FMS never calls SIO directly, but always goes through the disk interface vector because this is more convenient and saves precious ROM space.

The (undocumented) 1050 service commands are not supported by `DiskInterf`, and programs need to go through SIO to trigger them.

19 Os Service Routines

A couple of modifications and improvements have been made to miscellaneous Os service routines that are listed in the following.

19.1 SetIRQ

This service routine allows you to safely define an interrupt vector without running into the risk that the interrupt is triggered while the two-byte vector is inconsistent. If modifying an interrupt by hand, it may possibly happen that the interrupt is triggered right at the time when the first byte is modified but the second byte of the vector contains still the previous value. Unlike the original Os, however, the SetIRQ routine of Os++ also safely installs IRQ vectors by temporarily disabling interrupts, and neither uses the ANTIC `WSYNC` register. Hence, it will not interfere with DLI timing.

19.2 Memo-Pad, Self-test and Powerup-Display

Atari Os contains one vector that is called by the BASIC `BYE` command. For the old Atari series, this vector entered Memo-Pad mode, for the XL series it entered the self-test. The 1200XL introduced another vector that shows a power-up display if neither a disk nor a cartridge is found. Finally, the XL Os includes a vector to run the self test. For Os++, all three vectors point to the same ROM location that first uninstalls all reset-resident programs, in specific any custom DOS that has been loaded from disk, and initializes the ROM-based FMS++. It then enters the Dos command line. There is neither a Memo-Pad, nor a start-up display, nor a self-test.

19.3 Reset and Power-Up

Reset has been slightly modified to not include a tape boot anymore, i.e. the `START` button is no longer honored. `OPTION` however, is, and disables BASIC, as on the XL series ROM. Even though a RAM and ROM test is still performed on power-up, self-test is no longer present and cannot be entered in case a failure is found. Instead, the Os uses the following color-codes to indicate failures:

Table 8: Tape Modifiers

Background Color	Failure
Red	RAM Failure
Magenta	ROM Check-sum Failure
Blue	Unable to initialize start-up screen

19.4 Tape Support Functions

The Atari OS contained two tape support functions in its kernel that initializes SIO to send bytes to the tape, and that reads a block from tape. Since Os++ no longer includes tape support, these two functions only return errors. These two functions were only used internally for the tape bootstrap and should not be called by user code.

19.5 Parallel Bus Interface Support

Only very few devices ever made use of the parallel port interface Atari designed for its XL series, and documentation was hard to get. Due to ROM size constraints, Os++ does no longer include PBI support, and the PBI support functions in the Os kernel do nothing.

19.6 New Kernel Functions

Os++ includes a couple of new kernel support functions: `Init850Vector` at address `$E48F` makes the necessary SIO calls to bootstrap the 850 interface. Thus, no disk based program needs to be loaded to bootstrap the serial interface box. This function takes no parameters, returns with the carry flag set on error and with carry clear if everything went fine.

The vector `LaunchDosVector` at offset `$E492` launches the Dos command line. The DOS run vector at RAM offset `$0A` points by default to this location. It is still recommended to run the Dos command line through `DOSVECTOR`, though.

The vector `FMSInit` at offset `$E498` initializes FMS++. This vector is called by the power-up process if no bootable disk is found in the first disk drive, or the boot sector indicates that the disk contains an FMS++ boot sector. The `FMSINIT` vector first initializes the `D:` device, and checks whether the system is in a cold start or a warm start. In a cold start, it initializes the number of supported drives to two and the number of file buffers to three, and then first attempts to load `CONFIG.SYS`. Afterwards, drive and file buffers are initialized again as the number of disk drives and file buffers may have changed. Afterwards, it tries to load `HANDLERS.SYS` and `AUTORUN.SYS`, in this order. It then returns to the user. If `FMSINIT` detects that the `Esc` key has been pressed, loading the start-up files is disabled.

The vector `LaunchDupVector` at offset `$E49B` finally points to the DUP command line start-up vector. By default, the new RAM vector `DUPVECTOR` at location `$3F6` points to this location. While `LAUNCHDOS` goes through a full system reset, `LAUNCHDUP` is a secondary helper function that calls directly into the Dos command line. It should not be called directly. Instead, users may want to set `DUPVECTOR` to install a custom command line.

20 Changes in the MathPack

The MathPack is logically not part of the Os, but part of Atari BASIC. It offers mathematical functions for floating point support, elementary algebra, conversion functions between integer and floating point, and elementary support for transcendental functions.

Unlike the Os, the MathPack does not use a jump table, but uses fixed call-in addresses; thus, MathPack did not have, initially, any type of interface and was just part of the BASIC code.

The MathPack version that comes with Os++ has been enhanced to compute faster and more precise than the original code, though — due to constraint ROM space — is not as fast as other third party implementations. TurboBasic, for example, uses loop-unrolling to gain a lot of speed, but hence suffers from a larger memory footprint. The Os++ MathPack follows a better rounding policy and hence offers about one digit more precision than the original Atari implementation.

20.1 Information for Advanced Users

The MathPack uses a rather weird format for floating point representation, namely a BCD (binary coded decimal) format to the basis of 100. Each byte contains two decimal digits, and the exponent describes the relative position of the decimal dot as a byte offset. This format avoids some round-off errors when converting numbers to and from the human-readable form, but is otherwise slow to process for the machine. Especially multiplications and divisions suffer from this.

While Atari documented *most* MathPack functions, some additional functions were nevertheless used by third-party software and are implemented as well in Os++. In the following, all available MathPack functions are listed — addresses for the floating point memory registers are found in the memory map in section 21:

$D800 `AsciiToBCD` This function converts an ATASCII string in the buffer pointed to by `INBUFF` at offset `CIX` into a BCD number that is placed into the BCD register `FR0`. The conversion stops at the first non-recognized character, and if the string is a valid number, this function returns with the carry set cleared. On error, the carry flag is set.

$D8E6 `BCDToAscii` Does the reverse of the above and converts the floating point number in `FR0` into the math pack output buffer at `OUTBUFF`. Bit 7 of the last character in the string is set to indicate the end of the number.

$D8E9 `BCDToAsciiFlex` This is a new Os++ function that works like the above, but places its output into the buffer pointed to by `INBUFF` rather than using the fixed memory locations at `OUTBUFF`.

$D9AA `IntToBCD` Converts the two-byte unsigned integer in `FR0` and `FR0+1` into a floating point number in the BCD register `FR0`. Returns with carry cleared on success, which will always be the case.

$D9AD `IntXYToBCD` Similar to the above, but the integer is located in registers `X` and `Y`, where `X` contains the low-byte and `Y` the high-byte.

$D9D2 `BCDToInt` Converts the floating point value contained in the BCD register `FR0` into an integer in `FR0` and `FR0+1` if this is possible. This call rounds mathematically to the nearest integer and returns with the carry set cleared if a proper conversion was possible and the result fitted into two bytes. Otherwise, it returns with the carry flag set. This function returns its result also in the register pair `X` and `Y` where `X` contains the low-byte and `Y` the high-byte. Proper rounding and register return are extensions of the Os++ implementation.

$DA44 `ZeroFR0` Clears the floating point register `FR0`.

$DA46 `ZeroFRX` Clears the floating point register whose zero page address is in register `X`.

$DA48 `ZeroRgs` Clears `Y` bytes starting at the zero page address in register `X`. This call is not officially documented by Atari, but used by some programs.

$DA51 `LoadOutbuff` Loads the zero-page register pair `INBUFF` with the address of the MathPack output buffer area at `OUTBUFF`. This call is also not officially documented, but used by some programs.

$DA5A `TimesTwo` Multiplies the zero page register pair `ZTEMP4` and `ZTEMP4+1` by two, sets carry on overflow. This call is also not officially documented but used by some software.

$DA60 `BCDSub` Subtracts the number in the BCD register `FR1` from the value in the BCD register `FR0` and places the result back in `FR0`. Returns with the carry flag set on error, otherwise with carry cleared.

$DA66 `BCDAdd` Adds the number in the BCD register `FR1` to the contents of the BCD register `FR0` and returns the result of the addition in `FR0`. Returns with the carry flag set on error, or with carry cleared if the addition could be performed.

$DADB `BCDMul` Multiplies the contents of the BCD register `FR1` with the contents of the BCD register `FR0` and returns its result in `FR0`. If the multiplication did not overflow, returns with the carry flag cleared, otherwise returns with carry set.

$DB28 `BCDDiv` Divides the contents of the BCD register `FR0` through the value contained in the BCD register `FR1` and returns the result in `FR0`. If the division could be performed, returns with the carry flag cleared, otherwise with the carry flag set. The latter happens, for example, on a division by zero.

$DBA1 `SkipBlanks` Skips any blank spaces in the buffer pointed to by the zero-page pointer `INBUFF` at offset `CIX` and adjusts `CIX` accordingly to the index of the first non-blank character in this buffer. This function is used as initial step of `AsciiToBCD`, and also by some third-party software even though not officially documented by Atari.

$DBAF `TestDigit` Tests whether the character at the input buffer pointed to by `INBUFF` at offset `CIX` is a valid decimal digit. If so, returns with the numerical value of the digit in the accumulator and with a carry set clear. If the character is not a valid digit, returns with the carry flag set. Used by `AsciiToBCD` and some other third party software. Not officially documented by Atari, but present and in use.

$DC00 `Normalize` Normalizes the BCD register `FR0` to a valid BCD representation by adjusting the exponent and the mantissa in such a way that the first digit-pair of the mantissa is non-zero. Returns with carry set on error, or carry clear if the normalization succeeded. This call is not documented by Atari, but used by several third party products.

$DC04 `NormalizeExt` This call works like the above, but normalizes the two-digit extended version of `FR0` by also including a truncate to the non-extended version. Functions otherwise identical to the above by returning the result in `FR0` and carry cleared on success, carry set on error. This is an extended Os++ MathPack call not present in the original ROMs. The final processing step of many arithmetic functions use a second function to first normalize properly by this call, and then round to the limited precision of the BCD functions of the MathPack using a proper round-to-nearest step.

$DD40 `EvalPoly` Evaluates a polynomial with the Horner scheme. The coefficients of the polynomial are located in memory at the position pointed to by the register pair `X` and `Y`, with the low byte in `X` and the high-byte in `Y`. Coefficients are BCD-encoded numbers, six bytes each. The highest order coefficient is placed in the lowest memory address, all other coefficients in decreasing order in ascending memory locations. The number at which the polynomial is to be evaluated has to be placed in the BCD register `FR0`. The number of coefficients, i.e. the order of the polynomial plus one has to be passed in in the accumulator of the CPU. The output is provided in register `FR0`, and the carry is cleared if the evaluation succeeded; otherwise, the carry flag is set.

$DD89 `LoadFR0IndXY` Load the BCD register `FR0` with a BCD value in memory pointed to by the register pair `X` and `Y` where the `X` register contains the low and the `Y` register the high address.

$DD8D `LoadFR0IndPtr` Load the BCD register `FR0` with the BCD value in memory pointed to by the zero page pointer `FLPTR`.

$DD98 `LoadFR1IndXY` Load the BCD register `FR1` with a BCD value in memory pointed to by the register pair `X` and `Y` where the `X` register contains the low and the `Y` register the high address.

$DD9C `LoadFR1IndPtr` Load the BCD register `FR1` with the BCD value in memory pointed to by the zero page pointer `FLPTR`.

$DDA7 `StoreFr0IndXY` Store the BCD register `FR0` in memory at the address pointed to by the register pair `X` and `Y` where the `X` register contains the low-byte and the `Y` register the high-byte of the target address.

$DDAB `StoreFr0IndPtr` Store the BCD register `FR0` in memory at the address pointed to by `FLPTR`.

$DDB6 `Fr0ToFr1` Copies the value of the floating point register `FR0` into the floating point register `FR1`.

$DDC0 `BCDExp` Computes the exponential function to the basis of e of the argument in `FR0`. Returns the result in `FR0`. On error, sets the carry; clears the carry on success.

$DDCC `BCDPow10` Raises the basis ten to the power of `FR0`, returns the result in `FR0` and clears the carry on success. Returns with the carry flag set on error.

$DE95 `BCDFract` Computes the fractional function $(x - t)/(x + t)$ where the value of x is placed in the BCD register `FR0` and the value of t in the memory addresses pointed to by the register pair `X` and `Y`, with the low-byte in `X` and the high-byte in `Y`. All variables are encoded in BCD. The result is returned in `FR0`. Returns with the carry flag set on error, otherwise with carry cleared. Not officially documented by Atari, though used by the MathPack itself and some third party software.

$DECD `BCDLog` Computes the natural logarithm of the argument in `FR0` and returns the result in `FR0`. Sets the carry on error, or clears it on success.

$DED1 `BCDLog10` Computes the logarithm to the basis of ten of the argument passed in in `FR0` and returns the result in `FR0`. Sets the carry on error, clears the carry on success.

$DF6C `ONEHALF` Not a call-in function, but the six bytes at this ROM address contain the BCD constant for 0.5. Used by some third party software.

$DFAE `ATNPOLY` Contains eleven coefficients for a polynomial approximation of the arcus tangent in radiant. Used by BASIC. Programs should not depend on this as there are better (and more precise) methods to compute the arcus tangent than using this table.

$DFAE `NEARONE` Contains the BCD constant 0.9999999999, used by BASIC for the arcus tangent approximation. Also used by some third-party software.

$DFF0 `PIOVER4` Contains the ten-digit BCD approximation of the mathematical constant $\pi/4$. Used by BASIC and maybe some other software.

21 Memory Map

This section lists the memory map as used by Os++ and its applications. It is mostly identical to the memory map of the Atari XL operating systems. However, as the parallel port support and the tape support have been removed, memory locations used by these two subsystems are no longer in use and reserved for future expansions. If two locations are given, then the two variables together form a 16 bit value whose low-value is in the first byte and whose high-value is in the second byte. Memory addresses indicated by * are new to Os++ or have changed their meaning compared to the Atari XL operating system. Addresses indicated by ^{t} are system internal temporaries that cannot be modified to obtain an observable effect and that do not have any controlled value outside of the Os calls that require them. They should be left alone by user programs.

$00 * Unused, reserved.

$01 * Unused, reserved.

$02,$03 CASINIT If bit 1 of BOOTFLAG is set, the Os++ reset routine jumps through this vector to initialize a reset resident program. While CASINIT was originally set by the tape bootstrap routine, this function is no longer implemented and programs may want to use this vector to make programs reset resident. Note that CASINIT is called before DOSINIT, the FMS init vector.

$04,$05 RAMTEST,BOOTPTR[t] Used during the cold reboot RAM test and the disk bootstrap routine. Only for internal use.

$06 CARTFLAG Nonzero if the reset routine detected an inserted cartridge, zero otherwise. If set, the reset routine initializes the cartridge on reset and starts the cartridge. Otherwise, the DOSVECTOR is called to run the DOS user interface, let it be the command line or the menu.

$07 * Unused, reserved.

$08 WARMSTARTFLAG If nonzero, the current reset is a warm-start, otherwise it is a cold start. Application programs should use this flag to perform a full initialization on cold-starts. While any type of user data should be preserved on warm-starts, a full initialization is required on cold-starts. BASIC and other programming cartridges should use this flag to reset and initialize the user program buffer on a cold-start. Set by the Os, not modifiable by user programs.

$09 BOOTFLAG Defines which reset vectors to call on reset. If bit 0 is set, the Os reset routine calls through DOSINIT to initialize any type of disk boot. Note that Os++ does have a built-in FMS and does not require this vector to initialize the ROM-based FMS. If bit 1 is set, the reset routine calls through CASINIT. If both bits are set, CASINIT is called first, followed by DOSINIT. The Os automatically sets bit 0 after going through a successful disk bootstrap. Note again that the built-in FMS does not set this flag.

$0A,$0B DOSVECTOR Called from user programs to exit to the DOS user interface, either a command line or a menu.

$0C,$0D DOSINIT Called by the Os on reset if bit 0 of BOOTFLAG is set, typically as the result of booting a program from disk.

$0E,$0F APPMEMHI First byte of the high-memory area available for the operating system and not used by the application program. Application programs shall set this pointer to point to the lowest RAM location they do *not* require. RAM between MEMLO and APPMEMHI is allocated by the user application, RAM between APPMEMHI and MEMTOP can be claimed by both the user application and the operating system. The Os heap, specifically the screen buffer and the display list, start at MEMTOP, and the Os expands this buffer by potentially lowering MEMTOP until it reaches APPMEMHI. The application memory starts at MEMLO and ends at MEMTOP. Applications allocate memory from the Os and expand their heap by growing APPMEMHI until reaching MEMTOP.

$10 IRQSTATSHADOW Shadow register of the Pokey IRQSTAT register. This register controls which Pokey interrupts are enabled. All memory locations from IRQSTAT up to the end of the first half of the zero page at location $7F are initialized by the Os on reset, addresses from $00 through $0F are only initialized during a cold start.

$11 BREAKFLAG This flag is cleared whenever Pokey receives an interrupt generated by the BREAK key on the keyboard. Several Os++ functions initialize this flag to a non-zero value to be able to test for a break condition. Application programs may use this flag likewise: By initializing it to a non-zero value, a test for zero allows programs to check whether the BREAK key has been pressed since the last time the flag has been checked.

$12,$13,$14 CLOCK This is a three-byte counter that is incremented each vertical blank. This register is unusual as the highest-order byte is in the lowest register address at $12 and the highest address contains the least significant byte. CLOCK denotes memory location $14, with the higher-order bytes at CLOCK-1 and CLOCK-2.

$15 FMSTMP ^{t*} Temporary location used by the ROM-based FMS. Not to be touched by the user.

$16 DIRENTRYOFFSET ^{t*} Temporary used by the FMS for scanning file names. Not to be touched by the user.

$17 REDUCEDCMD * Temporary used by CIO. Contains the offset within the device command table to call for the currently executing CIO command. Not to be touched by the user. Note that Os++ uses this as a CIO temporary, though handlers may use this code to detect which function was called. They can not expect that this location is preserved over CIO calls.

$18 COMPONENTSTART ^{t*} Temporary used by the FMS, keeps the offset within the file specifications where the file name starts, i.e. the offset of the character behind the colon separating the device name from the file name. Internal use only, not to be touched by the user.

$19 FMSSTACK ^{t*} Keeps the stack pointer of the user calling into FMS to allow for a quick exit path on errors by restoring the S register from this location. Internal use only, not be touched by the user.

$1A FILECOUNTER ^{t*} If the user uses the FMS /n file name modifiers to disambiguate between several identical files, this location keeps the desired file index. If $FF, no file name count modifier has been used. Internal use only, not to be touched by the user.

$1B FREEDIRCODE ^{t*} FMS temporary, keeps the file index times four of the first free directory entry that could be used to create a new file. If odd, no free directory slot has been found yet. Internal use only, not to be touched by the user.

$1C * Unused, reserved.

$1D * Unused, reserved.

$1E * Unused, reserved.

$1F * Unused, reserved.

$20 ZINDEX Part of the CIO zero page copy of the current IOCB. This and all following zero page variables are initialized by CIO when called and are available within the device handler as fast reference. They become invalid as soon as CIO exits to the user. This variable is the offset of the currently used handler within HATABS of the currently used IOCB, or $FF if the currently used IOCB has not been opened to a handler.

$21 ZUNIT Part of the CIO zero page copy of the current IOCB. This is the unit number of the device of the currently active IOCB.

$22 ZCMD Currently active IOCB command.

$23 ZSTATUS Copy of the IOCB status. Not useful within device handler code, but the handler exit code is stored by CIO internally to this location before it is copied back to the IOCB.

$24,$25 ZADR Buffer address of the currently active IOCB.

$26,$27 ZPUT* Address of the put-one-byte vector of the currently active IOCB. Handlers may re-initialize this vector within its OPEN vector. CIO never uses this vector directly, but BASIC uses this vector to output characters to a channel. Application programs should *not* use this vector but call through CIO instead.

$28,$29 ZLEN Length of the buffer of the current IOCB command. This counter is updated by CIO, handler code may read these bytes, or — to implement burst IO — modify them accordingly.

$2A ZAUX1 Auxiliary data byte, used to define the channel mode for the channel OPEN command.

$2B ZAUX2 Second auxiliary data byte, also used by open to define details of the open command. Unused by most handlers, FMS++ uses this byte to distinguish between file and direct mode.

$2C,$2D ZHANDLERVEC Not a copy of the IOCB, but the pointer to the current device handler code. Device handlers that require ZAUX3 and above need to access the IOCB in page 3 directly.

$2E ZIOCB Channel number times 16 of the current IOCB.

$2F ZIOBYTE Used by CIO internally to store temporarily the byte to be written or the last byte read by the handler. Not to be modified by the device handler or the user.

$30 SERIALSTATUS^t Generated status by the interrupt layer of the SIO serial protocol. This and the following zero page addresses are used by the interrupt service routines of SIO and should not be touched by the user.

$31 SERIALCHKSUM^t Cumulative check-sum over the bytes transmitted over SIO so far.

$32,$33 SERBUF^t Serial input or output buffer pointer.

$34,$35 SERBUFEND^t Pointer to the first character behind the serial input or output buffer. Serial data is transmitted from or to the buffer pointed to by SERBUF up to, but not including the byte at SERBUFEND. Additionally, a check-sum is transmitted or expected, depending on the transmission direction.

$36 * Unused, reserved.

$37 * Unused, reserved.

$38 SERIALDATADONE^t Initially zero, set to non-zero as soon as the payload data has been received completely. Once non-zero, the check-sum is expected.

$39 SERIALXFERDONE^t Initially zero, set as soon as the serial data has been received completely, including the check-sum.

$3A SERIALSENTDONE^t Initially zero, set as soon as the serial data has been sent completely, including the check-sum.

$3B SERIALCHKSUMDONE^t Initially zero, set as soon as the check-sum has been sent.

$3C SERIALNOCHKSUM^t If nonzero, the received data does not include a check-sum and SIO does not expect to receive a check-sum.

$3D TAPEBUFFERPTR^t Not used by Os++ and reserved. Used by the disk-based tape handler. This stores the offset within the tape buffer to the next available byte in the tape buffer.

$3E GAPTYPE^t Not used by Os++ and reserved. Used by the disk-based tape handler. If zero, long inter-record gaps are written, if non-zero, short inter-record gaps are written instead.

$3F TAPEEOFFLAG^t Not used by Os++ and reserved. Used by the disk-based tape handler and negative if the EOF has been reached while reading bytes from the tape.

$40 RECORDSIZE ^{t*} Not used by Os++ and reserved. Used by Os++ to store the size of a tape record. This is 128 for the regular tape mode and 0 for the turbo tape mode, indicating 256 byte records.

$41 SERIALSOUND Non-zero if SIO should generate sound to allow the user to monitor the serial transmission. Zero to disable the sound. Even if this byte is zero, some serial sound may be heart because external devices may feed the audio input line of the SIO interface.

$42 CRITICIO If non-zero, the system vertical blank is shortened to prepare the system for a critical I/O operation. If so, only the CLOCK registers and the system timer zero are updated. Furthermore, the attract-mode is still enabled, and the GTIA GPRIOR mode is updated on some graphic modes. All other operations, specifically all other system timers, VBI shadow register updates, joystick and keyboard delay timers, paddle and stick shadow register updates are bypassed. This allows a quicker reaction to incoming serial data and avoids disturbance of the serial protocol.

$43,$44 FMSPTR^t Temporary pointer register for FMS internal use. Not modifiable by the user.

$45,$46 DISKBUFFER^{t*} Pointer used by FMS++ to store the current disk buffer keeping the VTOC of the currently active drive. FMS internal use only.

$47,$48 FILEBUFFER^t FMS++ pointer to the currently active file buffer. This buffer stores bytes read or written by the user until enough data is buffered to write them to disk, or until all data is read and the buffer needs refill from disk. FMS internal use only.

$49 * Unused, reserved.

$4A * Unused, reserved.

$4B * Unused, reserved.

$4C SCREENERROR^t Status register of the screen handler. This register keeps the status output transmitted back to CIO. This is not to be modified by the user.

$4D ATTRACT Incremented along with CLOCK-1, this register enables the screen saver, the *attract mode*, as soon as it reaches 128. If so, the screen changes periodically its colors. The register is reset by keyboard activity. User programs that do not expect keyboard input but want to disable the screen saver should reset this register frequently.

$4E DARKMASK A mask by which screen colors are logically and-ed before the color shadow registers are written to the hardware registers. In the regular mode, set to $FE, but with active screen saver instead set to $F6 to disable light colors.

$4F COLORMASK A mask which is exclusively or'd with the values in the color shadow registers before masking them and writing them to the hardware registers. Zero in the regular mode, this register is frequently updated if the screen saver is active.

$50 Unused, reserved.

$51 ANTICMODE^t Temporary register for the screen handler keeping the Antic hardware mode during the screen setup. Not to be modified by the user.

$52 LEFTMARGIN This is a user-modifiable control register of the editor handler that defines how many columns are kept free at the left edge of the screen. This register is initialized to two, meaning that the left two columns of the text console are not used by the editor handler.

$53 RIGHTMARGIN Rightmost column to be used by the editor in the console and text windows. This register is initialized to 39, meaning that the editor may use all columns of the regular text mode. The user may modify this register to change the behavior of the editor.

$54 CURSORROW Y-position of the screen input or output location, also Y-location of the text cursor of the editor handler.

$55,$56 CURSORCOLUMN X-position of the screen input or output location, also X-position of the text cursor of the editor. Automatically incremented when reading or writing bytes to the screen, then wrapped around at the end of the line. Also updated by the editor handler to reflect cursor movements.

$57 GFXMODE Stores the currently active graphics mode of the screen handler. Mode zero is the text mode used by the editor. The presence of a text window in graphical modes is *not* reflected by this variable, i.e. its value is always between zero and 15.

$58,$59 GFXORIGIN Stores the address of the left topmost address of the screen memory and thus the first byte of the screen buffer.

$5A OLDCURSORROW Keeps the previous X location of the screen cursor before updating it. Required as origin position for the screen handler line and area fill commands.

$5B,$5C OLDCURSORCOLUMN Keeps the previous Y location of the screen cursor. Similar to the above used by the line and fill operations.

$5D CHRUNDERCURSOR^{t} Keeps the ANTIC code of the character under the text cursor. Updated by the editor handler whenever the text cursor is drawn, and used when removing the cursor.

$5E,$5F CURSORPTR^{t*} Memory address where the cursor is currently drawn. The screen and editor handler remove the cursor then by writing CHRUNDERCURSOR to this address. This pointer is 0 if the cursor is currently invisible.

**$60*** Unused, reserved for the keyboard handler.

**$61*** Unused, reserved for the keyboard handler.

$62 PALNTSCSHADOW Identifies the TV norm the system is working with. Zero indicates an NTSC/60Hz system, one a PAL/50Hz system.

$63 LOGICALCOLUMN^{t} Offset from the start of the logical column to the current input or output position within a logical editor line. A logical editor line can be up to three physical lines long, and hence as long as 120 characters. This is the offset from the left edge of the first physical line to the current position in the line.

$64,$65 SCREENPTR^{t} Temporary pointer used by the screen handler used for various operations.

$66 STMP1^{t*} Temporary for the screen handler.

$67 * Unused, reserved for the screen handler.

$68,$69 DLBOTTOM, SCROLLPTR^{t} Used by the screen handler as a temporary during display list built-up, and by the editor as a temporary for scrolling.

$6A RAMTOP High-byte of the first non-usable memory location. The memory page indicated by RAMTOP is either the first page of the cartridge, the first page of the Os ROM or the first page of a memory hole if the system is not fully equipped with 48K (or 64K) of memory. Memory below this location is allocated by the screen handler.

$6B BUFFERCNT ^{t}Number of characters contained in the current logical line of the editor handler.

$6C STARTLOGICALROW^{t*} Y position of the start of the current logical line. Editor handler internal use only.

$6D STARTLOGICALCOLUMN^{t*} X position of the start of the current logical line. Editor handler internal use only.

$6E BITMASK^t Temporary variable of the editor used to locate and set tabulator stops.

$6F SHIFTMASK^t Temporary variable of the screen handler to read or write the color at the current cursor position.

$70,$71 ROWACCU^t Y error accumulator for the Bresenham line drawing algorithm of the screen handler.

$72,$73 COLUMNACCU^t X error accumulator of the Bresenham line drawer of the screen handler.

$74,$75 DELTAMAX^t Maximum of the row and column difference for line drawing and by that the length of the line in pixels.

$76 DELTAROW^t Absolute row difference between start and end point of a line to be drawn.

$77,$78 DELTACOLUMN^t Absolute column difference between line start and end column.

$79,$7A KEYDEF Address of the keyboard definition table, used by the keyboard handler. This table can be defined by the user to change the keyboard layout.

$7B SWAPFLAG^t This private flag of the editor handler indicates whether the cursor positions within the text window have been swapped into the editor cursor position globals. If negative, the current cursor position is within the editor window. Otherwise, it is a cursor position within the graphical window of the screen handler.

$7C * Unused, reserved for the keyboard handler.

$7D INSDAT^t Temporary used by the editor handler. Temporary for character insertion, also temporary to control the insertion of logical or physical lines.

$7E,$7F COUNTER^t Temporary of the screen handler, counts the number of pixels of the current line still to be drawn.

$80 to $ff The upper half of the zero page is reserved for application programs. One of the application programs that is contained in the ROM area of the system, though not part of the Os is the MathPack. The math pack uses memory locations from $D4 to $FF.

$D4 . . . $D9 FR0 Floating point register zero. This register contains a BCD coded floating point number with an exponent to the base of 100. It contains the primary input and the output for floating point operations. Sometimes FR0 and FR0+1 is used as a two-byte unsigned integer number.

$DA,$DB FR0EXT *^{t} Four additional guard digits to improve the precision of the computation. The guard digits are used as additional overflow and rounding space before reducing the output of a floating point operation back to the regular math pack precision.

$DC . . . $DF Unused, reserved.

$E0 . . . $E5 FR1 Floating point register one. This contains the secondary input, i.e. the second operand of binary arithmetic operations. Also used as an internal temporary for some operations.

$E6 . . . $EB FR2 Floating point register two. Used as temporary register for some internal operations.

$EC FRX ^{t} Used internally by the ASCII to floating point conversion to store the position of the exponent in the input.

$ED EEXP ^{t} Used internally as a temporary storage for various math pack operations. Typically used as a temporary storage for an exponent value.

$EE NSIGN ^{t} Temporary storage of a floating point sign flag for ASCII to floating point conversion and multiplication/division operations.

$EF ESIGN ^{t} Temporary storage of the exponent sign for the ASCII to floating point conversion, also temporary counter in the polynomial evaluation.

$F0 FCHFLG ^{t} Temporary storage for various flags in the ASCII to floating point conversion routine, also temporary storage in the exponent function.

$F1 DIGRT ^{t} Temporary storage for the number of digits to the right of the decimal dot in the ASCII to floating point and floating point to ASCII conversion routines. Also temporary flag in the exponential function.

$F2 CIX Offset into the math pack input buffer. This is an offset to the currently parsed character within the input buffer pointed to by INBUFF.

$F3,$F4 INBUFF Points to the input buffer of the ASCII to BCD conversion. CIX is the offset of the first character within this buffer to be converted.

$F5,$F6 ZTEMP1 ^{t} Temporary storage for various use in the MathPack functions.

$F7,$F8 ZTEMP4 Only used by the MathPack function TimesTwo, this 16 bit variable keeps a 16-bit integer that is doubled by the mentioned function.

$F9,$FA * Unused, reserved.

$FB Unused, reserved.

$FC,$FD FLPTR ^{t} Temporary zero page pointer used by the floating point load and store functions.

$FE,$FF FLPT2 ^{t} Temporary zero page pointer used by the polynomial and rational function evaluation operations.

$100 . . . $1FF This is the hardware stack of the 6502 and not used by the operating system for any other purpose.

$200,$201 VECDLI Pointer to the display list interrupt service routine. Called by the operating system when a display list interrupt was detected as the source of a non-maskable interrupt. It is in the responsibility of the user to save and restore CPU registers, including the accumulator, and resetting the decimal flag if required.

$202,$203 PROCEEDVEC Called by the Os when PIA generated an interrupt due to an active edge transition on the PROCEED input of the SIO connector. Currently not used by the operating system or any device handler. The Os saves the accumulator that must be restored before exiting from the interrupt, and the Os also clears the decimal flag.

$204,$205 INTERRUPTVEC Called by the Os upon detection of an interrupt from the SIO INTERRUPT input. Currently unused. Otherwise, the protocol is identical to the above.

$206,$207 BRKVEC Called by the Os whenever a 6502 software interrupt, i.e. a BRK instruction, is executed. The protocol is similar to the above, i.e. the accumulator has already been pushed onto the stack and the decimal flag is cleared. The Os implementation does nothing, i.e. just returns to the user code. Note that the BRK instruction increments the program counter by two, even though the instruction itself is only one byte long.

$208,$209 KEYVEC Called whenever an interrupt due to keyboard input has been received. The protocol is identical to the above. The Os default implementation stores the received key-code from Pokey, checks for keyboard de-bouncing and handles some elementary keyboard functions already in the interrupt handler, such as the console start-stop flag and the HELP key.

$20A,$20B SERINVEC Called by the operating system as soon as a byte has been received completely through the serial port and the Pokey input register is filled. Again, the accumulator is already pushed on the stack and the decimal flag cleared. The Os default implementation checks for the Pokey status, detects framing and overrun errors and stores data in the serial input buffer or tests for the correctness of the check-sum.

$20C,$20D SEROUTVEC Called by the Os whenever the output register of Pokey is is able to take new input and the next output byte to be transmitted can be placed in the output register.

$20E,$20F SERXMTVEC Called by the Os as soon as Pokey completed its serial output and the last bit of the output register has been transmitted. The Pokey serial output register is dual-buffered, with one user-accessible register that keeps the next character to be transmitted, and one internal temporary register that is shifted out of the serial port on each serial clock transition. SEROUTVEC is called whenever the user register got copied into the temporary serial output shift register and it is ready to take another input. SERXMTVEC is called when the serial shift register becomes empty, too, and hence the serial transmission is complete. Os++ also uses the SERXMTVEC initiate a new serial output transmission. This means that a user can overwrite this interrupt vector to fetch all, including the first byte, from a RAM location under the Os ROM. It also means that the purpose of the SERXMTVEC interrupt is also to initiate a serial transmission if SERIALCHKSUMDONE and SERIALSENTDONE are both zero.

$210,$211 POKEYTIMER1VEC Called whenever the first Pokey timer interrupt is triggered by a timer 1 under-run and its interrupt is enabled. The accumulator has already been rescued on the stack and the decimal flag is clear when this vector is called. The first Pokey timer is controlled by Pokey channel zero.

$212,$213 POKEYTIMER2VEC Called on a Pokey timer 2 under-run interrupt, controlled by Pokey channel one.

$214,$215 POKEYTIMER2VEC Called on a Pokey timer 4 under-run interrupt, controlled by Pokey channel three. Pokey channel three cannot create interrupts.

$216,$217 IMMEDIATEIRQVEC Called by the Os on any mask-able interrupt detection. By default, this vector points to the Os interrupt handler which saves the accumulator on the stack, detects the source of the interrupt and forwards handling to one of the interrupt vectors defined above. When this vector is called, almost no interrupt handling has taken place, only the decimal flag has been cleared. Specifically, the accumulator has not be saved on the stack.

$218,$219 VBITIMER0 Counter of the first vertical blank system timer. The system keeps five timers and counts their counter registers down every vertical blank. As soon as the counter reaches the value zero, the corresponding timer vector is called, or a timer flag is set. System timer 0 is the first of these timers, and this is its counter. The timer vector to be called on under-runs is VECVBITIMER0. Timer zero is special in the sense that it is also count down when CRITICIO is set. It is also used internally by SIO.

$21A,$21B VBITIMER1 Counter of the second vertical blank system timer. Count down every VBI, the system calls VECVBITIMER1 if this counter under-runs.

$21C,$21D VBITIMER2 Counter of the third vertical blank system timer. This timer only sets the flag at TIMER3FLAG on under-runs, it does not call a vector.

$21E,$21F VBITIMER3 Counter of the fourth vertical blank system timer. This timer sets the flag at TIMER4FLAG on under-runs.

$220,$221 VBITIMER4 Counter of the fifth vertical blank system timer. This timer sets the flag at TIMER5FLAG on under-runs.

$222,$223 VECIMMEDIATE Immediate system vertical blank handler. Whenever the system detects a vertical blank interrupt, it clears the decimal flag, saves all registers to the stack and calls through this vector. By default, it points to the Os vertical blank handler which copies RAM shadow registers to the hardware registers, handles the system timers, handles keyboard de-bouncing and copies joystick and paddle data to the shadow registers. The system vertical blank then calls VECDEFERRED discussed next.

$224,$225 VECDEFERRED Deferred vertical blank interrupt. Called by the system vertical blank handler as soon as it is done. By default, this just restores the registers from the stack and returns from the interrupt. Users may add additional service routines here. Note that the deferred vertical blank is not called if CRITICIO is set.

$226,$227 VECVBITIMER0 Vector of the first VBI system timer. Whenever the timer at VBITIMER0 under-runs, this vector is called. The called function should then potentially reload the counter and return to the system by an RTS.

$228,$229 VECVBITIMER1 Vector of the second VBI system timer, called whenever the timer at VBITIMER1 under-runs.

$22A TIMER2FLAG Set to non-zero as soon as timer 3 at VBITIMER2 under-runs.

$22B KEYTIMER This is an Os internal timer that is responsible for keyboard auto-repeat. As soon as a key-press is detected, this timer is re-initialized from KEYREPEATDELAY. If it becomes zero and the current key is still pressed, additional key presses will be generated. This auto-repeat function uses this very same timer, but initializes it with the value from KEYREPEAT.

$22C TIMER3FLAG Set to non-zero as soon as timer 4 at VBITIMER3 under-runs.

$22D IRQTEMP ^{t} Temporary storage for the SetIRQ Os function.

$22E TIMER4FLAG Set to non-zero as soon as timer 5 at VBITIMER4 under-runs.

$22F DMACTRLSHADOW Shadow register of the Antic DMACTRL register. Copied to the hardware register on each vertical blank.

$230 DLISTSHADOW Shadow register of the Antic DLIST register keeping the start address of the display list. Copied to the hardware register on every vertical blank.

$232 SKSTATSHADOW Shadow register of the Pokey SKSTAT register controlling the operations of the serial port. Only initialized, updated and controlled by SIO, and the disk-based tape handler.

$233 * Unused, reserved.

$234 PENHSHADOW Shadow register of the horizontal position of the light pen. Updated every vertical blank from the antic PENH register.

$235 PENVSHADOW Shadow register of the vertical light pen position.

$236,$237 BREAKVEC Called by the Os whenever an interrupt triggered by the BREAK key has been detected. The protocol is similar to the above, i.e. the accumulator has already been pushed onto the stack and the decimal flag is cleared. The Os implementation clears BREAKFLAG, the console hold flag STARTSTOPFLAG, re-enables the cursor and stops the attract (screen saver) mode.

$238 * Unused, reserved.

$239 * Unused, reserved.

$23A . . . $23D SIOCMDFRAME ^{t} These four memory addresses buffer the SIO command frame constructed from the SIO Device Control Block to initiate a SIO transfer. Serial output transmits then this command frame over the SIO hardware interface.

$23E SIOACK ^{t} Stores the command acknowledge returned from a SIO device upon receiving a SIO command frame.

$23F SIOERROR ^{t} Temporary storage for the SIO error code computed from SIOACK.

$240 DISKBOOTFLAG ^{t} First byte of the first sector of a disk used for disk bootstrap. This flag does not have a specific meaning and is ignored.

$241 DISKBOOTSECTORS ^{t} Second byte of the boot sector during disk bootstrap. This byte contains the number of sectors to load as part of the bootstrap function.

$242,$243 DISKBOOTADDRESS ^{t} Address into which the disk boot is to be bootstrapped. This is a copy from the third and fourth byte of the boot sector and used by the ROM bootstrap function.

$244 COLDSTARTFLAG Kept set during disk bootstrap, this flag turns a warm start into a full cold start, thus indicating that disk booting has not yet been completed and should be re-tried if interrupted. Can also be set by the user to force a cold start on the next system reset. Cleared as soon as the disk bootstrap is complete and the user booted function could initialize itself correctly. If the user provided bootstrap function does not return to the Os, the user should rather clear this flag to indicate that disk boot is complete. Some cartridges, specifically Mac/65, use this flag (incorrectly) to detect that their initialization vector has been called during a cold-start, and hence need to fully initialize themselves. Instead, the WARMSTARTFLAG should be used as this flag is only set as a side effect of the Os bootstrap function.

$245 * Unused, reserved.

$246 DISKTIMEOUT Used by the resident disk service vector, namely DiskInterf, this memory address contains the disk time out for formatting. It defaults to 160 seconds, but disk stations can set the value by transmitting the desired timeout with the third byte of the disk status return data. DiskInterf uses a hard-coded timeout of seven seconds for all other (non-formatting) commands.

$247 to $24B * Unused, reserved.

$24E,$24F VECNMI * This vector is called as soon as the NMI input line of ANTIC is pulled low. On the old Atari 800 and 400 models, this line was connected to the RESET console switch, and the NMI interrupt of the Os jumped into the Warmstart vector of the Os. On the XL systems, the pin remains open but can be connected to an external switch triggering this interrupt. While the XL operating system ignores this interrupt, it is vectored by Os++. By default, it points to the warmstart vector.

$24F to $269 * Unused, reserved.

$26A GPRIORSET * If this memory address is non-zero, the system vertical blank updates GPRIOR from its shadow register at GPRIORSHADOW even if CRITICIO is set. This is necessary if one of the GTIA graphics modes is used with a text window.

$26B * Unused, reserved.

$26C FINESCROLL ^{t} Internal counter for fine scrolling. This keeps the number of video lines the character cells are currently scrolled. The timer is updated in the vertical blank and initialized by the editor handler.

$26D KEYDISABLE If set, keyboard interrupts and keyboard auto-repeat are disabled, in other words: The keyboard is disfunctional.

$26E FINESCROLLFLAG If negative, fine scrolling is enabled. To use fine scrolling, a negative value shall be written into this register and the editor or graphics screen needs to be re-opened. This enables fine scrolling in the full editor, or in the text window of a graphics screen. Clearing this flag and re-opening the screen or editor disables fine scrolling again.

$26F GPRIORSHADOW Shadow register of the GTIA GPRIOR register. This register controls the player/missile priorities, but also the GTIA graphics modes. The hardware register is updated from the shadow register on regular vertical blanks, or on short vertical blanks if GPRIORSET is non-zero.

$270 PADDLE0 Shadow register of the position of the first paddle (if connected).

$271 PADDLE1 Shadow register of the position of the second paddle.

$272 PADDLE2 Shadow register of the position of the third paddle.

$273 PADDLE3 Shadow register of the position of the fourth paddle.

$274 PADDLE4 Originally, shadow register of the fifth paddle, however now a copy of PADDLE0.

$275 PADDLE5 Copy of PADDLE1.

$276 PADDLE6 Copy of PADDLE2.

$277 PADDLE7 Copy of PADDLE3.

$278 STICK0 Shadow register of the position of the first joystick. The joystick direction is encoded in bits zero to three.

$279 STICK1 Shadow register of the position of the second joystick.

$27A STICK2 Originally, shadow register of the position of the third joystick, now a copy of STICK0.

$27B STICK3 Copy of STICK1.

$27C PTRIG0 Shadow register of the state of the first paddle button. If zero, the button is pressed, otherwise it is released.

$27D PTRIG1 Shadow register of the state of the second paddle button.

$27E PTRIG2 State of the third paddle button.

$27F PTRIG3 State of the fourth paddle button.

$280 PTRIG4 Originally, state of the fifth paddle button, but now a copy of PTRIG0.

$281 PTRIG5 Copy of PTRIG1.

$282 PTRIG6 Copy of PTRIG2.

$283 PTRIG7 Copy of PTRIG3.

$284 STRIG0 Shadow register of the state of the button of the first joystick. If set, the button is released. If zero, the button is pressed.

$285 STRIG1 Shadow register of the state of the button of the second joystick.

$286 STRIG2 Originally, state of the third joystick button, but now a copy of STRIG0.

$287 STRIG3 Copy of STRIG1.

$288 * Unused, reserved.

$289 TAPEMODE * Unused by Os++. Used by the disk-based tape handler. If negative, indicates that the tape output buffer contains dirty bytes and requires flushing to tape when closing the stream.

$28A TAPERECORDLEN * Unused by Os++. Used by the disk-based tape handler, this memory address stores the number of valid bytes in the last read tape record that can be read by the user.

$28B ... $28F * Unused, reserved.

$290 WINDOWROW Y position of the editor cursor in the text window of a graphics screen. Graphics cursor position and editor cursor position are swapped between this and CURSORROW when SWAPFLAG is set. Then this location keeps a backup of the graphics cursor position, whereas CURSORROW keeps the position of the cursor in the text window.

$291,$292 WINDOWCOLUMN X position of the editor cursor in the text window of a graphics screen. Potentially the backup of CURSORCOLUMN if SWAPFLAG is set.

$293 WINDOWGFXMODE Graphics mode of the text window, thus zero, or backup of GFXMODE if SWAPFLAG is set.

$294,$295 WINDOWORIGIN Pointer to the memory address that keeps the left, top character of the text window; or backup of GFXORIGIN if SWAPFLAG is set.

$296 WINDOWOLDCURSORROW Previous Y location of the cursor in the text window, or backup of OLDCURSORROW if SWAPFLAG is set.

$297,$298 WINDOWOLDCURSORCOLUMN This address contains the previous X location of the cursor in the text window, or a backup of OLDCURSORCOLUMN if SWAPFLAG is set.

$299 WINDOWCHRUNDERCURSOR^t Keeps the ANTIC code of the character under the text cursor in the text window, or backup of CHRUNDERCURSOR if SWAPFLAG is set.

$29A,$29B WINDOWCURSORPTR ^{t} Memory address of the cursor in the text window, or backup of CURSORPTR if SWAPFLAG is set.

$29C SIOCMDRETRY ^{t} Retry counter for the SIO command frame transmission. This counter counts the number of retries to submit a SIO command frame. Initially set to 13, the counter is decremented on each failed attempt to transmit a SIO command until SIO finally gives up. The maximum number of retries is hard-coded and not user-selectable.

$29D MAXROWS ^{t} Temporary of the editor, keeps the number of lines to scroll if a logical line moves out of the screen or text window.

$29E * Unused, reserved.

$29F * Unused, reserved.

$2A0 PIXELMASK ^{t} The value in this address is a bit mask that has bits set at the location where data is written to the screen or read from the screen. This temporary is used by the screen handler to write data to the screen.

$2A1 * Unused, reserved.

$2A2 ESCFLAG If negative, the last character input or output was an ESC $1B, and the next following control character is to be printed on the screen literally instead of being interpreted by the editor handler. EOL $9B, however, remains a special case and always advances to the next line, regardless of this flag.

$2A3 . . . $2B1 TABSTOPS This is a 120 bit mask that has a bit set wherever in a logical line a TAB stop exists. The TAB sequence or the TAB key advances to the next TAB stop recorded in this array.

$2B2 . . . $2B5 LINESTARTS A 32 bit mask that has a bit set whenever a physical line is also the start of a logical line. The last byte of this table is actually initialized when building the screen, but never used as the text screen is only 24 lines high.

$2B6 INVERSEMASK Set to 128 if the inverse video key was pressed and all alphanumerical characters input by the keyboard should be output in inverse video. Set to zero for regular input.

$2B7 FILLFLAG^t Temporary of the screen line drawer. Set if the current line drawing operation is actually part of an area fill operation and the area to the right of the line shall be filled until detecting a non-zero pixel.

$2B8 OVERRUNFLAG^{t*} If non-zero, the editor detected that the character insertion is close to overrunning the logical line, and the buzzer shall sound.

$2B9 * Unused, reserved.

$2BA * Unused, reserved.

$2BB SCROLLFLAG^t Temporary of the editor, this flag counts the number of lines scrolled out of the screen to adjust the cursor position.

$2BC* Unused, reserved.

$2BD SIORETRY^{t*} Number of times minus one a complete SIO transaction shall be retried when an error is received. This is initially set to one, indicating that each SIO operation should be at least tried twice. The initial value of this counter cannot be adjusted and is hard-coded.

$2BE SHIFTLOCK Reflects the state of the CAPS keyboard function. Upon regular input, this memory address is zero. If set to $40, the CAPS key has been pressed and small print characters can be entered. If set to $80, CTRL and CAPS were pressed, converting all input keys into control sequences.

$2BF WINDOWHEIGHT Size of the editor text window. On a regular text screen, the value in here is 24 indicating that the text editor has 24 lines available. If a graphics mode with a text window is in use, the value in here is 4, indicating that only four lines are available for the editor. The Os++ editor also accepts values between 24 and 4 here to split a text screen into an upper static part and a lower window part that scrolls. This window part need not to be four lines high, but may have an arbitrary height.

$2C0 PCOLOR0SHADOW Shadow register of the player 0 color. Copied into the hardware register on every vertical blank.

$2C1 PCOLOR1SHADOW Shadow register of the player 1 color.

$2C2 PCOLOR2SHADOW Shadow register of the player 2 color.

$2C3 PCOLOR3SHADOW Shadow register of the player 3 color.

$2C4 COLOR0SHADOW Shadow register of the playfield 0 color.

$2C5 COLOR1SHADOW Shadow register of the playfield 1 color.

$2C6 COLOR2SHADOW Shadow register of the playfield 2 color.

$2C7 COLOR3SHADOW Shadow register of the playfield 3 color.

$2C8 COLORBACKSHADOW Shadow register of the background color.

$2C9 . . . $2D4 * Unused, reserved.

$2D5,$2D6 DISKSECTORSZ Size of a disk sector for disk access through DiskInterf. This size is copied into the disk control block when DiskInterf calls SIO.

$2D7 * Unused, reserved.

$2D8 * Unused, reserved.

$2D9 KEYREPEATDELAY Number of vertical blanks a key must be pressed before the keyboard auto repeat function is activated.

$2DA KEYREPEAT Number of vertical blanks between each simulated key press once the keyboard auto repeat function is activated.

$2DB NOCLICK If non-zero, the keyboard click generation is disabled and the keyboard handler does not create an audible feedback when typing.

$2DC HELPFLAG This flag is set whenever the HELP key has been pressed. Applications should reset this flag frequently and test it for non-zero values to detect whether the user requested help. A regular HELP key generates the value $11, if help is pressed together with Shift, the value $51 is kept here, HELP with Control generates $91.

$2DD * Unused, reserved.

$2DE NEXTPRINTERIDX *t* Number of characters in the printer output buffer.

$2DF PRINTERBUFSIZE Size of the printer output buffer in characters. This is 40 for the regular print mode, 20 for double-wide character printing, and 29 for sideways printing.

$2E0,$2E1 RUNVECTOR Binary load files store here the run vector where the program is supposed to be started. The run vector is typically initialized by the binary load file itself by indicating a load address that includes this vector.

$2E2,$2E3 INITVECTOR Binary load initialization vector, jumped to after each block loaded from disk, and reset once used. Binary load programs may store here an address to be jumped through by the FMS scatter loader. This happens by including a binary load block that includes the init vector.

$2E4 RAMSIZE Filled by the Operating system in the system initialization phase by the number of RAM pages available for the Os and applications. This is in its meaning and value identical to RAMTOP, except that RAMTOP is initialized by the screen handler from RAMSIZE. The value does not change later on.

$2E5,$2E6 MEMTOP Pointer to the first byte in the high-memory area used by the operating system heap not available for application programs. Application programs may use the memory range between MEMLO and MEMTOP and shall set APPMEMHI to indicate to the Os the highest address they require.

$2E7,$2E8 MEMLO Pointer to the first byte available for application programs not required by the operating system. Without the FMS, the first free application byte is $700, but with the FMS, additional memory is required for the disk buffers and the FMS globals.

$2E9 * Unused, reserved.

$2EA . . . $2ED DISKSTATUS When requesting the disk status with the S command through DiskInterf, the four-byte response from the disk drive is stored in these bytes. The first byte contains the status report of the disk firmware, the second byte the status of the disk controller. The third byte selects the time out for the format command in seconds, the fourth byte is always zero.

$2EE,$2EF BAUDRATE The approximate baud rate of the tape recorder. Initialized by the Os to 300 baud, the disk-based tape handler adjusts this value when reading from tape by an approximation based on the sync bytes on the tape. The tape handler initializes or adjusts this flag, it is not user-controllable.

$2F0 CURSORINHIBIT If non-zero, the editor does not render the cursor and leaves the cursor invisible. This flag is reset by the default BREAK interrupt or by a warm-start.

$2F1 KEYDELAY Used by the keyboard de-bounce function. Count down in the system vertical blank, the keyboard interrupt does not react on the same key pressed again if this counter is non-zero.

$2F2 LASTKEY Keeps the keyboard code of the last key pressed, used to compare against the current key to potentially ignore the key press within the de-bounce period.

$2F3 CHCTRLSHADOW Shadow register of the Antic CHCTRL hardware register. This register controls how the characters are rendered in the text based Antic modes, and how characters with the highest bit set are rendered to the screen.

$2F4 CHBASESHADOW Shadow register of the Antic CHBASE register, base address of the character set for text-based modes.

$2F5 NEWCURSORROW Destination row, i.e. Y position, of a line draw or line fill operation of the screen handler.

$2F6,$2F7 NEWCURSORCOLUMN Destination column, i.e. X position, of the line to be drawn by the screen handler.

$2F8 DELTAROWSIGN ^{t} Sign of the row difference between source and destination row of the line to be drawn. This is $+1$ if the line is drawn to the right, or -1 if the line is to be drawn to the left.

$2F9 DELTACOLUMNSIGN ^{t} Sign of the column difference between source and destination column of a line to be drawn. $+1$ for drawing a line downwards, -1 for drawing it upwards.

$2FA SCREENCHAR ^{t} Internal ANTIC code of the last character written to or read from the screen. Used only as an internal storage of the screen handler.

$2FB SCREENBYTE ^{t} Storage of the input or output of the last screen handler CIO Get or Put operation.

$2FC KEYCODESHADOW Shadow register of the Pokey KEYCODE register, this memory address keeps the keyboard code of the last key press detected.

$2FD FILLCOLOR Pixel color to be used by the area fill operation of the screen handler.

$2FE DIRECTFLAG If set, control characters are not interpreted by the editor handler, but are written directly to screen as graphical characters. This flag has an effect similar to ESCFLAG, though is independent of it. Users may set this flag to disable the interpretation of control characters by the editor. The only special case, the EOL character, remains. This control sequence is always interpreted as a line feed.

$2FF STARTSTOPFLAG If set, any output to the screen or editor is inhibited and halted. The program flow then blocks in the screen handler output function. This flag is toggled by the keyboard interrupt handler when detecting the keyboard sequence Control 1, it is reset by the Os BREAK key interrupt vector or a system reset.

$300 SIODEVICEID This and the following twelve bytes define the Device Control Block by which SIO transmissions are defined. This memory cell contains the target device ID a SIO command addresses.

$301 SIODEVICEUNIT The device unit, from one up, used to distinguish between several units of the same device model, e.g. the disk drive unit.

$302 SIOCOMMAND The command to be executed by the SIO device.

$303 SIOSTATUS When initiating a SIO command, this memory location defines the data flow direction. Commands that transmit data from the device to the host have bit 6 set, commands that transmit data from the host to the device have bit 7 set. If neither bit 6 nor bit 7 are set, the command does not transmit any additional data and is consists only of the command frame. On return, it contains the SIO error code, a duplicate of the SIO return code transmitted in the Y register.

$304,$305 SIOBUFFER Points to the output buffer containing the output data to be transmitted or the input buffer taking the data read from the device.

$306 SIOTIMEOUT Timeout of the operation in seconds (approximately). If the device does not answer within this time period, SIO first tries to re-initiate the transmission but finally gives up returning a timeout error.

$307 Unused, reserved.

$308,$309 SIOSIZE Size of the input or output buffer in bytes.

$30A SIOAUX1 Additional data refining the SIO command to be transmitted as part of the SIO command frame. For disk drives, the sector low byte is stored here.

$30B SIOAUX2 Secondary additional data included in the SIO command frame. This contains the sector high-byte for disk drive commands, but is otherwise target device dependent.

$30C,$30D BAUDCNT ^{t} Unused by Os++, but used by the disk-based tape handler as temporaries for the baud rate computation.

$30E * Unused, reserved.

$30F * Unused, reserved.

$310 TAPEBLOAD ^{t*} Contains a temporary storage for the flags defining the binary load procedure of the disk-based tape handler.

$311 * Unused, reserved.

$312 Unused, reserved.

$313 Unused, reserved.

$314 PRINTERTIMEOUT Timeout for printer device commands submitted by the printer handler. Defaults to 30 seconds, but a printer can request a different timeout by returning a suitable timeout as third byte of the SIO device status command.

$315 Unused, reserved.

$316 SERIALINBIT ^{t} Unused by Os++, but used by the disk-based tape handler. This is a temporary storing the state of the serial input line. It is used for scanning for the tape sync markers.

$317 SIOTIMERFLAG ^{t} A flag maintained by SIO and cleared by a system timer zero under-run on a time-out by a custom timer vector installed by SIO. SIO then aborts the transmission and re-tries to access the device.

$318 SIOSTACKPTR ^{t} Used both by SIO and the screen handler to store the stack pointer to allow a quick exit path in case an error has been detected.

$319 SIOSTATUSTMP ^{t} Temporary storage for the SIO return code before it is moved to SIOSTATUS.

$31A ...$33C HATABS The CIO device handler table. Each entry consists of three bytes, the device name (letter), and a pointer to the device vector table defining the entry points into the device handler routines. The table is terminated by a zero device letter. This handler table allows to define up to ten CIO devices, five additional devices to the system defined devices E,S,K,P,C. While Os++ does not include a tape handler, it still defines a dummy C: device here that only returns errors and that is replaced by the disk-based tape handler.

$33D ...$33F INITMAGIC If these three bytes contain the special values $5c,$93,$25 on a CPU reset, the system is considered initialized and a warm reset returning the system into an orderly state is performed. Otherwise, the reset triggers a full cold start followed by disk bootstrap. The magic values do not have a special meaning except that it is assumed that the RAM chips do not contain by pure chance the mentioned special values when turning on the system.

$340 IOCB0Index This and the following sixteen bytes define the first IOCB, the CIO input/output control block. Such a control block can be understood as a file handle or I/O channel to a handler to perform IO-operations on a higher abstraction level. In total eight IOCBs exist, allowing applications to open at most eight I/O channels at once. Channel 0 is by default connected to the editor device allowing console input or output. This first byte of IOCB 0 is either $FF if the channel is currently closed, or an offset into HATABS where the handler specification is found.

$341 IOCB0UNIT The device unit, from one up, identifying one out of multiple devices handled by the same device handler. This unit may, for example, identify one out of several disk drives.

$342 IOCB0CMD Command to be issued by the device, or by CIO. Commands define whether a channel is to be opened, closed, whether a block or a record of bytes is to be read or written, whether a device status is to be received or whether any special I/O control operation is to be performed.

$343 IOCB0STATUS Error code of the last I/O operation performed over this channel.

$344,$345 IOCB0ADR Buffer address containing the data to be written, or collecting the data to be read.

$346,$347 IOCB0PUT Pointer into the byte-put routine of the handler. Not to be used, only exists for legacy reasons.

$348,$349 IOCB0LEN Length of the buffer, defines the maximum amount of data to be transmitted on a read or write command.

$34A IOCB0AUX1 First additional data of IOCB channel 0. This data is command specific, but defines for example the open mode when opening a channel to a device.

$34B IOCB0AUX2 Second additional data of IOCB channel 0. Again command specific, but for example also used by the open command to define an access mode.

$34C IOCB0AUX3 Third additional command specific data. For the FMS, this data contains the low-byte of the sector number for the POINT and NOTE commands.

$34D IOCB0AUX4 Fourth additional command specific data. This contains for example the high-byte of the sector number for the POINT and NOTE commands of the FMS.

$34E IOCB0AUX5 Fifth additional command specific data. The FMS POINT and NOTE commands use it to transmit a byte offset from the start of the sector.

$34F IOCB0AUX6 Sixth additional command specific data, currently not used by any handler.

$350 ...$35F IOCB1 IOCB control block 1. Its layout is identical to the above.

$360 ...$36F IOCB2 IOCB control block 2.

$370 . . . $37F `IOCB2` IOCB control block 3.

$380 . . . $38F `IOCB2` IOCB control block 4.

$390 . . . $39F `IOCB2` IOCB control block 5.

$3A0 . . . $3AF `IOCB2` IOCB control block 6.

$3B0 . . . $3BF `IOCB2` IOCB control block 7.

$3C0 . . . $3E7 `PRINTERBUFFER` Output buffer of the printer handler.

$3E8 `SUPERFLAG` If this flag is set, the keyboard handler signals to the editor handler that an extended keyboard function request has been received for which no ATASCII representation exists. Extended keyboard functions use otherwise regular ATASCII codes, but include functionalities such as moving the cursor to the home or end of screen position, to the left or right boundary that are otherwise not available. Extended keyboard functions can be defined by redefining the key-map.

$3E9 * Unused, reserved.

$3EA * Unused, reserved.

$3EB `CARTSUM` Check-sum of 256 bytes from `$BFF0` up, extending into the Os ROM area. The reset function of the Os uses this sum to test whether a cartridge has been changed. If so, a reset becomes a full cold-start.

$3EC * Unused, reserved.

$3ED . . . $3F4 Unused, reserved.

$3F5 `FMSBOOTFLAG` * This flag controls various FMS++ related functions during reset. If bit 0 is set, the reset routine signals application programs and cartridges that the application program area has been overwritten by signaling a cold-start. This is used whenever the Dos required application memory for its own purpose, e.g. as buffer for copy operations. If bit 4 is set, FMS++ will not allocate low-memory area for disk buffers. This flag can be set to relocate buffers by an extension, typically the FMS overlay manager. If bit 6 is set, the reset routine does not start application programs, run launches the Dos user interface through `DUPVECTOR`. If bit 7 is set, the FMS++ initialization is run by the Os reset routine. Note that `DOSINIT` is *not* used by FMS++ but remains available for application programs.

$3F6,$3F7 `DUPVECTOR` * Dos user interface start-up vector. This vector points to the immediate entry point of the Dos interface, usually the Dos command line. `DOSVECTOR` typically goes through a reset cycle to re-initialize the machine, then calls through `DUPVECTOR` to start the user interface. This vector points by default to the `LaunchDup` kernel function.

$3F8 `BASICDISABLED` If non-zero, the built-in BASIC ROM has been disabled by the user, and the reset routine will keep it disabled.

$3F9 Unused, reserved.

$3FA TRIGGER3SHADOW Shadow register of the GTIA `TRIG3` hardware register that is used to identify an inserted cartridge. This flag is compared against `TRIG3` on a reset and issues a cold-start if a cartridge change has been detected. The XL operating system furthermore compared it in its vertical blank interrupt and intentionally locked up the system by an endless loop in case its value changed. This feature is *no longer* included in Os++ as it caused several compatibility problems.

$3FB Unused, reserved.

$3FC Unused, reserved.

$3FD . . . $4FF * Unused by Os++, but used by the disk-based tape handler. Note that the Turbo mode of the tape handler writes 256 byte records and hence requires all of page four, and not only its lower half as in previous versions of the operating system, as its input and output buffer. If only the regular tape modes are used, the upper half of page 4 remains free.

$500 . . . $6FF `BOOTSPACE` * Used by Os++ for boot-strapping the 850 interface box if one is found as SIO device. Note that the disk-based 850 bootstrap code included in older versions of the operating system also used the very same memory region for 850 interface bootstrap, though never documented this.

$580 . . . $5DF `OUTBUFF` Output buffer of the math pack `BCDToAscii` conversion function. If the converted ASCII number should be placed in a different buffer, users may also load the buffer address into the zero-page pointer `INBUFF` and call `BCDToAscii+3` instead.

$5E0 . . . $5E5 `POLYBUFF` ^{t} Temporary floating point register used by the math pack `EvalPoly` function.

$5E6 . . . $5EB `EXPBUFF` ^{t} Temporary floating point register for the `BCDLog` and `BCDLog10` functions of the math pack.

$600 . . . $6FF Unused by Os++ and available for user applications, though neither part of the Os nor of the application heap.

$700 . . . $800 * If FMS++ is initialized by setting bit 7 of `FMSBOOTFLAG`, this page keeps FMS++ variables. FMS++ disk and file buffers start above this area, followed by the application program area.

$700 . . . $703 `RESETENTRY` * Unused by FMS++, but the FMS overlay manager stores its reset vector here.

$704 . . . $708 * Unused, reserved.

$709 `FMSBUFFERS` Number of file buffers FMS++ should allocate. This defines the number of files FMS++ can open simultaneously. By default, the value is three. A `CONFIG.SYS` file can install other values here.

$70A `FMSDRIVEMASK` Defines which disk drives are allocated for FMS++ by a bit-mask, where the LSB defines a flag for drive 0, the next bit for drive one and so on. This mask by that also defines the number of disk buffers FMS++ will allocate. By default, FMS++ handles drives 0 and 1, and thus requires two disk buffers. `CONFIG.SYS` can change this value.

$70B * Unused, reserved.

$70C,$70D `DISKBUFFERBASE` * Base address of the disk buffers. FMS++ keeps its disk buffers from this address up, 128 bytes per buffer.

$70E . . . $711 * Unused, reserved.

$712,$713 `FILEBUFFERBASE` * Base address of the file buffers, from this address up. 128 bytes per buffer.

$714 * Unused, reserved.

$715 . . . $728 `FILENAMEBUFFER` ^{*t} A short buffer for keeping a file name while scanning the directory.

$729 `BLOADFLAGS` ^{*t} An 8 bit flags mask that defines activities during binary load. This is an internal temporary.

$72A `BLOADIOCB` ^{*t} Keeps the IOCB number for the binary load operation as a temporary.

$72B,$72C `STARTADDRESS` ^{*t} A temporary that keeps the start address of a binary load block.

$72D,$72E `ENDADDRESS` ^{*t} A temporary that keeps the end address of a binary load block.

$72F * Unused, reserved.

$730 . . . $737 AVAILFLAGS *t One byte per drive that identifies whether this drive is available for FMS usage. This is an expanded version of FMSDRIVEMASK and only for internal FMS usage.

$738 . . . $73F FILEBUFFERFLAGS *t File buffer allocation flags, one byte per file buffer. If the corresponding flag is zero, the file buffer is available. Otherwise, it is in use.

$740 . . . $74B NEWDTAB * Unused by FMS++, but used by the FMS Overlay manager to define a replacement disk handler. The D: HATABS entry points here if the overlay manager is active. Otherwise, it points to the ROM area.

$74C . . . $754 REGULARENTRY * All handler entries of the replacement disk device handler installed by the overlay manager go here and call a dispatcher in the overlay management code. Dispatching into the specific handler code is then by a table indexed through REDUCEDCMD. The test for an active overlay manager shall be made by testing the first byte of this region to contain the value $20, and the byte at offset 6 to contain the value $4C.

$755 . . . $75D POBENTRY * The overlay manager put-one-byte vector is here. This also goes through a small dispatcher code before calling the actual overlay manager code and then, finally, the FMS++ code.

$75E . . . $769 SWITCHOFF * Utility code of the overlay manager to switch any cartridge off. This is cartridge specific, and hence this service code is cart-specific. Application programs that may want to make use of the overlay manager may jump in here to bank out the cartridge.

$76A . . . $775 SWITCHON * Utility code of the overlay manager to bank in the current cartridge. Again, the overlay manager prepares the proper code here and application programs may want to use it for banking. Note that this call and the above do *not* nest.

$776 . . . $778 * Unused, reserved.

$779 WRITECOMMAND Keeps the SIO command FMS++ uses for writing sectors to disk. Should be changed by the /V and /O file name extensions instead of changing this value here.

$77A . . . $77F * Unused, reserved.

$780 . . . $78F FCB0BASE *t This, and the rest of page seven contain FMS private extensions of the IOCB. Each IOCB has a correspondence here that extends the IOCB by 16 bytes of internal flags and values for FMS++ usage. The block from $780 up contains the extensions for IOCB 0, the next 16 byte block from $790 the extensions for IOCB 1 and so on.

$780 FILECODE0 *t This memory cell contains the file directory index times four if channel 0 is open to a file. Otherwise, the value in here is odd.

$781 FILEAUX10 *t Keeps the open mode for any file channel 0 is open for, that is, 4 for reading, 8 for writing and so on. This is a copy from IOCB0AUX1 made once the channel gets opened. Otherwise, its value is undefined.

$783 SPECIALFLAG0 *t Number of data bytes in a regular sector by a file or disk opened through channel 0. If this value is 128, the special *direct mode* is used that reads and writes to disk without a directory or file structure.

$782*t Unused, reserved.

$784 ACCESSFLAGS0 *t Defines management of the sector linkage. If bit 6 is set, the file buffer is dirty and requires updating to disk. If bit 7 is set, the sector links within the file buffer require updating and the file needs extension beyond its current bounds by reallocating a new sector from the VTOC and updating the linkage. This is again for any file opened on channel 0.

$785 BYTEMAX0 *t Number of valid data bytes within the current sector opened through channel 0. This is unlike SPECIALFLAG0 not the maximum number of bytes *available* in a sector, but the actual number of *valid* bytes within the sector read.

$786 BYTEPOSITION0 *t Byte offset within the current sector currently read or written, for a file opened through channel 0. This value can also be obtained through the CIO NOTE command.

$787 OPENBUFFER0 *t If not $FF, this is the file buffer index allocated for the file open on channel 0. Otherwise, i.e. if $FF, no file buffer has been allocated.

$788,$789 FILESECTOR0 *t Current sector index for the file opened through channel 0. This value can also be obtained through the CIO command NOTE.

$78A,$78B NEXTSECTOR0 *t In case the linkage to the next sector in the file is already known, its sector address is stored here. This value is extracted from the linkage bytes of the current file sector. This is again for a file opened through IOCB 0.

$78C,$78D *t Unused, reserved.

$78E,$78F FILELENGTH0 *t Keeps the size of the file in sectors. This value is read from the disk directory, and/or is updated when extending the file beyond its end.

$790 ... $79F FCB1BASE *t File control block 1. These 16 bytes are structured identically to those above, but apply to IOCB 1 instead of IOCB 0.

$7A0 ... $7AF FCB2BASE *t File control block for IOCB 2.

$7B0 ... $7BF FCB3BASE *t File control block for IOCB 3.

$7C0 ... $7CF FCB3BASE *t File control block for IOCB 4.

$7D0 ... $7DF FCB3BASE *t File control block for IOCB 5.

$7E0 ... $7EF FCB3BASE *t File control block for IOCB 6.

$7F0 ... $7FF FCB3BASE *t File control block for IOCB 7.

$800 FMSEND * End of FMS globals. If the FMS Overlay manager is not active, file and disk buffers start from here and extend upwards, 128 bytes per buffer. If the overlay manager is active, this is the start of the application program space and MEMLO points here.

$5000 DUPINIT * If the self-test ROM is banked in, this area contains the Dos command line. The same ROM area also contains parts of the Os initialization and bootstrap code that is not required once the system is running. The self test present in the ROMs of the XL series is no longer included, though a partial RAM and ROM test is included in the cold start procedure.

$8000 CART16KBASE 16K cartridges map here into the memory space of the 6502 and extend up to $BFFF.

$A000 CART8KBASE 8K cartridges map into here and extend to $BFFF.

$BFF0 CARTSUMREGION Start of the area upon which the cartridge check-sum is computed. This, and the next 256 bytes are summed up and form a check-sum that is compared upon a reset. In case the cart check-sum differs from the previous sum, a full cold start is initiated.

$BFFA,$BFFB CARTCRUN Start address of the cartridge. Once the Os initialized itself, CASINIT and DOSINIT vectors returned and the Dos user interface is not to be run, the cartridge is started through this vector.

$BFFC CARTTEST Must contain the value zero to allow the Os to recognize the cartridge. If nonzero, the cartridge is ignored.

$BFFD CARTTYPE Contains the cartridge flags. Bit zero defines whether a disk bootstrap process shall be initiated. If zero, no disk boot is attempted. Bit 2 is used to indicate whether the cartridge is to be started. If zero, the cartridge run address is not used, instead the Os reset function continues to launch the Dos user interface through DOSVECTOR. Bit 7 identifies a diagnostic cartridge. If set, CARTINIT is called immediately after initializing the CPU hardware stack. No further system initialization is performed. Otherwise, CARTINIT is called after initializing the hardware and the Os vectors, immediately before opening the text console.

$BFFE,$BFFF CARTINIT Cart initialization vector. This is called immediately before the text console is opened for regular cartridges, and very early directly after initializing the CPU for diagnostic cartridges.

$C000 . . . $C00B FMSTABLE The entire lower ROM area from $C000 to $CFFF is exclusively reserved for the FMS++ code. The FMS is initialized if the first disk drive is not found during bootstrap, or does not contain a valid boot sector, i.e. the boot load address is zero. The cold-start then sets bit 7 of FMSBOOTFLAG indicating that the Os reset routine should initialize the FMS. FMS initialization starts with buffer allocation, loading of CONFIG.SYS, then re-initialization again, followed by loading of HANDLERS.SYS and AUTORUN.SYS. Once complete, FMS bootstrap returns. If a cartridge is found, it is run. Otherwise, DOSVECTOR is called which sets bit 6 of FMSBOOTFLAG and resets the system again. This goes through a warm start system initialization which, finally, calls through DUPVECTOR. This starts the ROM code that banks in the Dos command line code into page $50 and starts execution there. The first 12 bytes of the lower ROM area contain the handler entry points for the FMS++ HATABS vector. The FMS ROM code extends up to $CBFF.

$CC00. . . $CFFF INTLCHARMAP Contains the international character set. It is slightly modified from the international character set coming with the original XL ROM because the character at ATASCII position 96 contains now the German sharp s which was missing from the original set.

$D000 . . . $D7FF Contains the hardware registers of the machine. Some areas are reserved for future extensions. The hardware registers are not documented here.

$D800 . . . $DFFF Contains the math pack service routines. These are actually part of the BASIC programming language, though might be used for other purposes. The math pack ROM and its changes for Os++ are described in section 20.1.

$E000 . . . $E3FF STDCHARMAP Regular character set used by the machine. The character map is sorted by internal ANTIC representation, eight bytes per character.

$E400 . . . $E40F EDITORTABLE The call-in vectors for the editor handler, pointed to by the E entry in HATABS, and the editor init vector at $E40C. The byte at $E40F is unused.

$E410 . . . $E41F SCREENTABLE The handler call-in vectors for the screen device S.

$E420 . . . $E42F KEYBOARDTABLE Handler vectors for the keyboard K device.

$E430 . . . $E43F PRINTERTABLE Handler vectors for the printer device P.

$E440 . . . $E44F TAPETABLE * Handler vectors for a dummy handler that only returns error codes. The tape device is no longer ROM-based but exists as a disk-based solution. The C device is initialized, but does not provide any function.

$E450 `DiskInitVector` This and the following addresses are part of the Os ROM kernel functions, a standardized set of functions at fixed and defined addresses to be used by application programs. While absolute addresses of ROM service routines did change in the past, the addresses of the kernel functions remained constant. This vector here initializes all necessary Os variables for a standard 1050 disk station, required by the next disk interface service vector. In specific, it defines the format command time out to 160 seconds and the sector size to 128 bytes.

$E453 `DiskInterfVector` The Os ROM disk interface vector. This vector sits one level above the serial interface vector `SIO` and keeps care of initializing the `SIO` RAM variables correctly for disk drive communications. In specific, it sets the command directions `SIOSTATUS`, `SIODEVICEID` to address the disk drives, `SIOTIMEOUT` and `SIOSIZE`. The caller only needs to provide the disk drive unit `SIODEVICEUNIT`, `SIOCOMMAND`, `SIOBUFFER` and `SIOAUX1`, `SIOAUX2`. The latter define the sector address to operate on. `DiskInterf` knows the 1050 read, write with and without verify commands, the `SIO` status command, and the single and enhanced density format commands. FMS++ jumps entirely through `DiskInterf` as it supports all disk drive functions, and not an incomplete subset as in earlier ROM versions. On a `SIO` status command, the disk format timeout is read from the drive, the status bytes are returned in `DISKSTATUS` and no user buffer need to be provided.

$E456 `CIOVector` The Os central I/O service function. Operates on IOCB, input-output control blocks, and performs byte, block or record-oriented I/O functions. Opens and closes channels. Os++ CIO is compatible to the original CIO except that `REDUCEDCMD` is initialized and the put one byte vector of the IOCB is filled by a default value *before* the handler open vector is called, hence leaving the handler the chance to modify this vector before returning to the ROM. `CIO` is the Os top-level I/O function and the one to be used by application programs.

$E459 `SIOVector` The Os serial I/O service function, responsible for any type of I/O operation going through the serial bus. This vector performs the low-level communication to external devices on the serial bus and is typically used by the Os handlers, such as the printer device or FMS++. The device to address and command to perform are submitted in `SIODEVICEID` and up, in page three. The `SIO` in Os++ is only capable to communicate with "intelligent" bus devices, it cannot talk to the tape recorder. Tape recorder functionalities have been removed due to constraint ROM space and have been relocated into a disk-based handler that includes both the missing `SIO` functions and the missing `CIO` functions.

$E45C `SetIRQVector` This function is capable to safely install a two-byte interrupt vector without running into a race condition of an incompletely installed vector while the interrupt is triggered. The accumulator contains the vector index to define, where zero identifies the `IMMEDIATEIRQVEC` at address `$216` and every increment of `A` increases the modified vector address by two. In specific, vector six is the immediate and vector seven the deferred vertical blank vector. Vectors eight and nine are the system timer zero and one call-in address. Unlike the original ROM function, this function does not have a race condition for installing mask-able interrupts, it does block them correctly, and this version neither touches the Antic `WSYNC` register, hence does not cause hick-ups in the display timing. The `X` register contains the high-byte, and the `Y` register the low-byte of the register to be set.

$E45F `ImmediateVBIVector` Starts the Os immediate VBI. If an application program installs its own custom immediate VBI, it should leave it by calling the Os VBI through this vector.

$E462 `ExitVBIVector` Performs the register cleanup required to leave the deferred VBI. An application program that changes the deferred VBI vector should exit its own VBI through this vector to perform the necessary cleanup activity and to safely return from the VBI interrupt.

$E465 `SIOInitVector` Initializes the PIA and Pokey hardware registers and shadow registers to allow serial port usage. Is called by the ROM reset routine.

$E468 * Internal ROM vector not be used. This vector initializes the hardware registers for a serial port output operation, but instead SIO should be used to trigger any type of serial port activity. Not used by the ROMs directly.

$E46B `NMIInitVector` Initializes the NMI enable register of AMTIC, and updates the shadow register of `TRIG3` to allow cart change detection. This vector does so little that there is actually no point in using it.

$E46E `CIOInitVector` Initializes the IOCB registers by setting all of them forcefully to the closed status, without actually closing any of them. Should not be called by application code, but is used by the Os during system initialization.

$E471 `ByeVector` * Used by the `BASIC BYE` command to leave the cartridge. On Os/A, the memo-pad mode was started. For the XL-Roms, this starts the self-test. For Os++, any disk-booted software is removed, the Os memory globals are re-initialized, the ROM-based FMS++ is initialized and the system exits to the Dos command line. Thus, `BYE` leaves a disk-based Dos and exits safely to the ROM-based FMS++.

$E474 `WarmStartVector` Re-initializes the system by going through a simulated warm-start. This resets all hardware devices, re-initializes most Os vectors, re-opens the text console, initializes the tape and disk boots and finally exits either to the cartridge or the Dos command line.

$E477 `ColdStartVector` Re-initializes the system completely by going through a simulated cold-start. In addition to the above, the application memory is cleared, the low and high memory pointers are reset clearing the operating system and application heap, and disk-boot is attempted as if the machine has just been turned on.

$E47A * Obsolete and no longer supported. This was an internal Os function used by the tape handler and the Os bootstrap code to load a block from tape.

$E47D * Also obsolete and no longer supported. This started a tape read operation and was both used by the tape device and the Os tape bootstrap. As both are no longer existent, this vector is replaced a function that only returns an error code.

$E480 `PowerupDisplayVector` * Originally used by the 1200 XL ROMs to show a power-up display if no cartridge is inserted, this vector has been replaced in the XL ROMs to be identical to the `ByeVector`. This remains the case for Os++ except that `ByeVector` itself changed and re-initializes FMS++ and exits to the Dos command line.

$E483 `SelfTest` * Originally used by the XL ROMs to initiate the self test, this vector has always been identical to the `ByeVector`. This did not change in Os++, except that `ByeVector` now re-initializes FMS++ and launches the Dos command line.

$E486 `MountHandler` Installs a new device into the `HATABS` table of the Os. On entry, the `X` register contains the device letter, the accumulator the high, and the `Y` register the low-byte of the handler service table to be installed. The function exits with the carry cleared if the operation was performed, and the carry set if the operation could not be performed. In the latter case the CPU negative `N` flag is set if `HATABS` is overflowing. It is cleared if the call failed because a handler of the same device letter is already present. In the latter case, the accumulator and the `Y` register are preserved, whereas `X` contains the offset from `HATABS` to the conflicting vector, *not* the device letter entry. The caller may then replace the conflicting handler by its own, or may abort to install the handler. That is, upon return `HATABS,X` is the low, and `HATABS+1,X` the high-address of the conflict.

$E489 * Obsolete parallel port service routine. As the PBI interface is not supported by Os++, this function no longer exists and only returns an error code.

$E48C * Obsolete parallel port service function.

$E48F `Init850Vector` * Performs all necessary bootstrap activity to load the 850 serial port interface box handler code into the ROM, i.e. the `R:` device. No `AUTORUN.SYS` file is required for this, a simple `RUN E48F` command in the Dos command line is sufficient. The `HANDLERS.SYS` file on the Os++ system disk does exactly that.

$E492 `LaunchDosVector` * This is where the `DOSVECTOR` RAM variable points to by default. This vector calls the Dos command line of the ROM-based Dos.

$E495,$E496 `DUPBUFFER`* Not a vector, but a pointer to the Dos command line input buffer. This buffer contains the full command line entered by the user. Dos-callable applications may want to parse command line options from this buffer. Note that Os++ does not provide a service routine to parse off command line arguments, unfortunately.

$E497 * Unused, is zero.

$E498 `FMSInitVector` * Init function of FMS++. If no bootable disk is found, the reset routine of the Os initializes FMS++ through this vector.

$E49B `LaunchDupVector`* Entry point of the Dos command line. The RAM vector `DUPVECTOR` points here. This vector should not be directly called by application programs as it does not go through a system initialization. Instead, the top-level `DOSVECTOR` should be called. Instead, application programs may want to replace the Dos command line by installing a custom command processor at `DUPVECTOR`. This vector points by default to this ROM location, assuming that all necessary system initialization has been done.

$E49E . . . $E4BF Unused and reserved for future extensions.

$E4C0 `RTSVector` This is a single `RTS` instruction that terminates the kernel ROM area. It is guaranteed to be here, and application programs in the need of a "dummy function" may use it.

$E4C1 . . . $FFF9 This is the high-area of the Os ROM, and contains the `CIO`, `SIO`, screen, editor, keyboard and printer device and miscellaneous service routines. The functions in the high-area ROM area shall not be called directly.

$FFFA,$FFFB `NMIENTRY` Defined by the 6502 CPU, this vector contains the entry point of the 6502 non-maskable interrupt service routine. The Os++ NMI first checks whether the interrupt was caused by a display list interrupt and then calls `VECDLI`. If the interrupt source is a vertical blank interrupt, registers are saved on the stack and the decimal flag is cleared, then `VECIMMEDIATE` is called to start the immediate vertical blank interrupt service routine. If the interrupt was caused by the ANTIC interrupt in pin, `WarmstartVector` is called. If the interrupt source could not be determined, the CPU returns.

$FFFC,$FFFD `CPURESET` Defined by the 6502 CPU, this vector is called when the CPU detects that the reset pin is pulled low. The Os then first disables all interrupts and enters a wait loop to wait for power becoming stable. It then checks the RAM variables `INITMAGIC`. If they have the correct values, a warm start is performed. Otherwise, the system goes through a complete cold start.

$FFFE,$FFFF `IRQENTRY` Also defined by the 6502 CPU, this vector is used to define the mask-able interrupt vector. The Os IRQ vector only clears the decimal flag and then jumps through `IMMEDIATEIRQVEC`. This Os function then determines the source of the interrupt and branches to the corresponding interrupt service routine.